THE UNIVERSITY OF QUEENSLAND

AUSTRALIA

An Ontology-Based Simulation Assembly Infrastructure

William Waterson

A thesis submitted for the degree of Doctor of Philosophy at

The University of Queensland in September 2011

The Department of Chemical Engineering

Declaration by Author

This thesis is composed of my original work, and contains no material previously published or written by another person except where due reference has been made in the text. I have clearly stated the contribution by others to jointly-authored works that I have included in my thesis.

I have clearly stated the contribution of others to my thesis as a whole, including statistical assistance, survey design, data analysis, significant technical procedures, professional editorial advice, and any other original research work used or reported in my thesis. The content of my thesis is the result of work I have carried out since the commencement of my research higher degree candidature and does not include a substantial part of work that has been submitted to qualify for the award of any other degree or diploma in any university or other tertiary institution. I have clearly stated which parts of my thesis, if any, have been submitted to qualify for another award.

I acknowledge that an electronic copy of my thesis must be lodged with the University Library and, subject to the General Award Rules of The University of Queensland, immediately made available for research and study in accordance with the *Copyright Act 1968*.

I acknowledge that copyright of all material contained in my thesis resides with the copyright holder(s) of that material.

Statement of Contributions to Jointly Authored Works Contained in the Thesis

No jointly-authored works.

Statement of Contributions by Others to the Thesis as a Whole

No contributions by others.

Statement of Parts of the Thesis Submitted to Qualify for the Award of Another Degree

None.

Published Works by the Author Incorporated into the Thesis

None.

Additional Published Works by the Author Relevant to the Thesis but not Forming Part of it

None.

Acknowledgements

I would like to thank my supervisor, Professor Ian Cameron for his input throughout the development of this thesis. I would also like to thank Associate Professor Roger Duke for his proofreading of software-related technical content in this document.

I would like to thank Professor Paul Lant and Catherine Crawford for their recommended minor changes to the overall thesis and its content in the month just prior to the final 2011 submission.

Lastly, I would like to thank my family and friends for their assistance, patience, and understanding over many years. This thesis would not exist had it not been for their support.

Abstract

In this thesis we analyse, design, specify and implement an ontology-based simulation-assembly infrastructure, to more intuitively direct the task of simulation assembly. For real-world problems, the task of simulation assembly requires a skill set well beyond most simulation practitioners. Real world simulation problems are complex, large scale and overwhelming. Related software tools supporting simulation practitioners are often written by yet other simulation practitioners who are not software engineers themselves.

By dividing the roles involved in simulation creation—according to skill level and expertise—into practitioner, developer and architect, we rationalise how to partition the functionality of the overall software system and related workflow. We found that it was more intuitive to use 3D layers for visualising the conceptual assembly of simulation problems. The implementation demonstrates the feasibility of presenting complex concept sets consisting of any number of concepts in a form accessible to the end user to (a) navigate and create concept instances in each layer and (b) to provide extensibility of the environment to arbitrary ontologies. We found that the building block mathematical techniques underpinning the resulting system must have an accompanying visual definition to configure those techniques within the practitioner's runtime graphical environment. We also found that recent technologies now enable the implementation of key components within the infrastructure, in particular the interoperability of mathematical techniques written in multiple programming languages. The solution of using the intermediate language representation (IR) facilitated by the LLVM Compiler Infrastructure alleviates the issue of selecting a canonical programming language for use by developers to implement their mathematical techniques. This permits developers to implement their mathematical techniques in arbitrary, commodity programming languages.

The advantages of this approach are that simulation practitioners (1) can work within their existing skillset to assemble simulations, (2) don't have to be literate in text-based programming languages, (3) have access to more ontologies to configure their simulation environment to any problem domain, (4) won't have to deal with software engineering issues in their workflow, all leading to (5) a higher confidence in the results produced from simulation.

Keywords

ontology, simulation, directed, acyclic, layers, concept, technique, software, process engineering.

Australian and New Zealand Standard Research Classifications (ANZSRC)

080603 Conceptual Modelling 40%, 080602 Computer-Human Interaction 30%, 090407 Process Control and Simulation 30%.

Contents

Chapter 1: Introduction

The task of assembling simulations for real-world modelling problems requires a skill set beyond most practitioners. In this thesis, practitioners are considered to be domain experts in a particular field of study other than classical computer science or information technology—for example process engineering, chemistry, physics, or biochemistry. Implicitly, simulation practitioners originate from the actual field to which they apply modelling to obtain useful predictions. As domain experts in that particular field, their primary skillset involves an understanding of the phenomena, concepts, and abstract mathematical techniques applicable to their discipline. Literacy in coding text-based programming languages is a secondary or peripheral skill—hereafter, the phrase "to code" shall mean "to write text-based source code". Furthermore, it is even less likely that a given practitioner will understand the processes necessary to applying these programming languages in a structured manner necessary to maximising the probability of creating reliable, bugfree simulations for generating trustworthy predictions.

To compound matters, often the software tools supporting simulation practitioners are written by yet other simulation practitioners, who are not themselves software engineers. The resulting software tools consequently suffer from the problems described above, and are also highly specialised for the intended domain of the practitioner, preventing the reapplication of canonical aspects of such software to yet other problem domains and simulation problems.

Real world simulation problems are complex, large scale and overwhelming. Practitioners must deal with a multitude of interacting phenomena and mechanisms that underpin nontrivial modelling studies. The modern practitioner must simultaneous cope with this aspect of the problem, while balancing the task of assembling the simulation itself, a combination that weighs upon the practitioner's cognitive resources. This loading distracts the practitioner from their primary task and focuses them away from the intended task of producing and analysing reliable simulation predictions.

The results of simulation are often used to provide an informed decision regarding real-world design, whose consequences in many cases risk significant costly resources, capital, and reputations. If the exercise of simulation is to produce reliable predictions that can be used confidently by decision makers, then we must consider better ways that maximise the simulation practitioner's primary abilities—rather than distract them with peripheral demands that are secondary or peripheral to their primary task.

In this thesis we consider something better. This work divides the roles involved in simulation assembly according to skill level and expertise of the parties involved. These roles are practitioner, developer and architect. Based on this segregation of roles, we rationalise a better partitioning of functionality of the overall software system. Consequently, each role can contribute to the overall workflow of simulation creation at their particular level of expertise and specialisation.

This thesis analyses, designs, specifies and implements an ontology-based simulation-assembly infrastructure around this partitioning of roles. From the Merriam-Webster Dictionery, there are two definitions for the term "ontology", one being a branch of metaphysics concerned with the nature and relations of being, the other a computer science term relating to a categorisation of concepts and connecting relationships, as used by the practitioners of various disciplines to comprehend problems within their domain. This thesis is concerned with the latter definition.

The thesis shows that it is more intuitive to present 3D layers for visually displaying the conceptual assembly of simulation problems to the practitioner. This approach can display any number of concept types in the practitioner's simulation assembly environment—unlike traditional 2D block-diagram flowsheeting. The approach enables the practitioner to navigate these layers and create instances of the concept types specific to each layer, and then to subsequently connect the relationships amongst these instances between layers to express the overall simulation problem. The approach also provides extensibility: the practitioner can access more ontologies to configure their simulation-assembly environment to any problem domain—aided by the flexibility of the 3D layered, visualisation approach.

We found that the building block mathematical techniques underpinning the resulting system must have an accompanying visual definition to configure those techniques within the practitioner's runtime graphical environment. We also found that recent technologies now enable the implementation of key components within the infrastructure, in particular the interoperability of mathematical techniques written in multiple programming languages. The solution of using the

intermediate language representation (IR) facilitated by the LLVM Compiler Infrastructure alleviates the issue of selecting a canonical programming language for use by developers to implement their mathematical techniques. This permits developers to implement their mathematical techniques in arbitrary, commodity programming languages.

The advantage of this approach is that simulation practitioners can work within their existing skillset to assemble simulations. The practitioner does not have to be literate in text-based programming languages using the approach, because they visually configure building blocks then set the parameters of those building blocks prior to simulation solution. Elimination of text-based coding also removes software development tasks from the practitioner's workflow, so they need not deal with software engineering issues—such as code design, performance optimisation, debugging, testing, maintenance.

1.1 Document History and Online Artifacts

This work was completed and submitted in 2003. Due to particular circumstances, it is only being resubmitted now—September 2011—with the approval of The University of Queensland.

The entirety of this thesis has been placed into the public domain at http://archive.org and http://www.williamwaterson.com.

The source code for the software implementation presented later in this thesis is downloadable from https://github.com/williamwaterson/protolayer.

1.2 Intended Audience and Technical Level

This thesis has been carefully structured to be accessible to simulation practitioners unfamiliar with software design nor originating from computer science. Much of this work builds upon basic software design theory explained clearly yet concisely in early chapters. The work proceeds by describing many fundamental terms, then gradually building upon them as the thesis progresses. As such, the work is approachable to both practitioners and software developers.

1.3 Originality and Contribution

Thesis demonstrates that we can analyse, design, and implement an ontology-based simulation-assembly infrastructure around the separation of roles of practitioner, developer, and architect. It's originality stems from demonstrating how to partition the resulting software system into respective components so that each of role can work within their primary skillset, to reliably contribute to the process of assembling simulations. In this way, we develop a system that shows:

- Practitioners do not have to be literate in text-based programming languages—a secondary or peripheral skill to these specialists—to assemble simulations.

- Practitioners can have access to more ontologies so as to reconfigure and extend their simulation-assembly environment, should that environment be immediately inadequate to their particular problem domain.

- Practitioners can be removed from software development process involved in writing reliable and tested mathematical techniques, lowering the cognitive burden upon the practitioner by allowing them to configure existing building blocks within their environment.

In this way, we can maximise the confidence in the results produced from simulation.

Further contributions of this work are as follows. We found that it was more intuitive to use 3D layers for visualising the conceptual assembly of simulation problems. The implementation demonstrates the feasibility of presenting complex concept sets consisting of any number of concepts in a form accessible to the practitioner to (a) navigate and create concept instances in each layer and (b) to provide extensibility of the environment to arbitrary ontologies.

We found that the building block mathematical techniques underpinning the resulting system must have an accompanying visual definition of an associated dialogbox, to configure those techniques associated with concept instances within the practitioner's runtime graphical environment.

We found that recent technologies now enable the greater interoperability of mathematical techniques written in multiple programming languages. The thesis has rationalised that the future incorporation of the intermediate language representation (IR) facilitated by the LLVM Compiler Infrastructure—see http://llvm.org—will alleviate the issue of selecting a canonical programming language for use by developers to implement their mathematical techniques. This would permit

developers to implement their mathematical techniques in arbitrary, commodity programming languages, storing the resulting universally interoperable LLVM IR code from these languages in the mathematical-technique repositories discussed in this work.

1.4 Document Structure

This document consists of nine chapters that discuss and develop the software infrastructure through analysis, design, specification, and implementation.

In Chapter 2, technical background is provided explaining software development terminology, particularly notions of statically-typed object-oriented software design and programming, as applied in later chapters to develop the ontology-based simulation assembly infrastructure. The chapter also performs a review of literature regarding both general simulation software and computer-aided process-engineering software.

Chapter 3 applies software analysis and software design to the workflow of simulation assembly, according to the roles of practitioner, developer and architect. The chapter extracts necessary requirements for the desired system, and then designs the software model for the infrastructure around these requirements.

Chapter 4 and 5 present respectively a specification for the infrastructure based on this design, and how this design was implemented in the present study.

To demonstrate how the software system is used, Chapters 6, 7 and 8 provide respectively how an example ontology is analysed in the context of the infrastructure's software model, the formal specification of this example ontology in UML, and a demonstration walkthrough in using the software to assemble simulations based on this ontology.

Chapter 9 concludes with a summary of the project outcomes, recommendations, and future work.

1.5 Summary

In this chapter we have introduced the rationale behind the present thesis in its goal to analyse, design, and implement an ontology-based simulation-assembly infrastructure. In the next chapter we begin with the development of background knowledge in the area of software design theory for the practitioner/reader, and provide a literature review of existing simulation software approaches.

Chapter 2: Background and Literature Review

Tools suited to simulation assembly, within any given discipline, draw upon technologies from computer science and information technology. This chapter lays the groundwork for the rest of the study by viewing such tools from this broader perspective. Important knowledge regarding basic software theory is spelled out for those unfamiliar with the notions that shape software development. The chapter focuses strongly on understanding object-oriented design and implementation within the statically-typed object-oriented programming languages used later in this work to implement the software infrastructure. The chapter also discusses certain information technology ideas currently gaining popularity within the simulation community. Subsequently, the study narrows its perspective to that of simulation software in general, and finally to process engineering simulation in particular. Along the way, the chapter critiques existing literature regarding simulation assembly from the above basis, and questions existing software effectiveness in addressing these tasks.

2.1 Terminology

The disciplines of mathematical simulation and software development each involve different terms and practises. The practises and terminology of a modelling practitioner are usually unfamiliar to a software developer, and vis-versa. It is important that the respective roles, tasks, and abstractions be clearly delineated, prior to actual analysis, because a reader approaching this text is likely to belong to only one of these camps. The goal is to avoid confusion, particularly for terms common to both fields. For this reason, the remainder of this document shall apply terms based on the following definitions. Certain names have been selected arbitrarily to make delineation easier. Note also that certain roles will be redefined in the next chapter.

From the physicochemical domain, the following terms arise:

Process: A collective assembly of states combined with influential mechanisms that drive changes in those states.

Model: The conceptual description of the mechanisms and entities composing a process, combined with selected mathematical techniques to solve this description. A model may evolve through several levels of complexity before adequately predicting process behaviour, within realistic error tolerances.

Simulation: An implementation of this model through the use of software.

Result: The final simulation selected through model refinement. The result also encompasses data generated by the simulation, as well as recommendations arising from the simulation predictions for improving the related process.

Two significant roles also require definition:

Practitioner: A person skilled in the art of modelling, who applies their abilities to develop simulations of actual processes. A practitioner can assume several roles, a primary task being the identification of mechanisms relevant to a model. While the following document suggests that a practitioner should not indulge in the tasks of programming and software implementation, the work does recognise that current practitioners engage in these tasks.

Developer: A person skilled in the tasks of software analysis, prototyping, design, and implementation. The developer creates tools, reusable libraries, graphical-user interfaces, and applications that automate the tasks of the practitioner. Software development should be considered separate to that of modelling.

From the realms of computer science and information technology, the following software-development terms can be defined:

Analysis: The software-development term for the examination of a problem domain and the related work processes targeted for improvement through the introduction of software tools. Analysis identifies the tasks and abstractions specific to that problem domain.

Requirements: A formal set of characteristics expected from the final software. These requirements arise by analysing the problem domain to be addressed. The requirements might be thought of as the conclusions arrived at through analysis.

Specification: A formal document outlining the software architecture for addressing the requirements. The specification expresses the data types, and data model, expected by the software's end user. Portions of a specification may be left open-ended, to be implemented in a manner chosen by the developer. Note the stress upon the specification being independent of any implementing technology: it is effectively the set of rules and constraints governing behaviour.

Prototype: Similar to an implementation, except that a prototype provides an experiment in code and design, to crystallise the developer's thoughts in identifying (a) the requirement set and (b) a specification that can be feasibly implemented. Prototyping provides a means of analysis that, like a case study, leads to realistic software requirements. One could call prototyping a form of bootstrapping for analysis and requirements. Prototypes are never intended for widespread distribution, as they are neither complete nor well engineered. The reason? A sizable part of the production of software products involves maintenance through managed software-development processes.

Implementation: An expression of the specification as carefully engineered software. Implementation involves the writing of code that adheres to coding conventions, built and tested under software-engineering principles, to minimise the number of bugs, maximise performance, and provide continued improvement through feedback from users, all according to quality-assurance processes. To meet these conditions, the software is measured against software metrics to analyse various quality characteristics. Note that the choice of implementation technology is secondary to the abstract specification of the software. One should consider that several different technology approaches could be used to implement the one specification. Hence, the presentation of an implementation without any independent specification could be considered myopic: an arbitrary choice in implementation prior to any independent reasoning.

2.2 Software Basics

Existing literature regarding simulation addresses issues of simulation expression, simulation solution, and proposed architectures, coding languages, and applications implemented to achieve these tasks. The following sections summarise this literature, examining both general simulation and simulation within the field of process engineering. Much work in the area of simulation uses methods and technologies adopted from computer science and information technology. It follows that much of a critique of simulation-assembly software be based in those terms. For this reason, an overview of basic software concepts is presented prior to this review, to aid unfamiliar readers.

These basics also clarify reasoning behind choices made in subsequent chapters, and justify the direction taken by the present study. The reader may wish to skip to the next section and come back for reference purposes. However, while these basics may seem trivial, their neglect affects the correct identification of software requirements for simulation-assembly software, because the work processes for simulation expression have traditionally involved programming. The key features to note, due to their reoccurrence throughout this study, are (a) object-oriented software methodologies—section 2.2.2—(b) domain ontologies in relation to traditional knowledge-based systems—section 2.2.3—and (c) how the duplicated efforts of the software-engineering and artificial-intelligence communities have created problems in software development for the process-simulation community—also section 2.2.3.

2.2.1 Basic Programming Concepts

The classic approach to implementing software is through writing code in a programming language. Code is stored in text files that are collectively termed the *source code* or *source program*. To process the commands expressed in the source code, the related files are scanned to recognise lexical patterns within the text. Expressions within the text are parsed into an abstract syntax tree (AST), to check that the text conforms to the grammatical rules of the language. If well formed, the internal representation is then either *compiled* or *interpreted*, depending on the specific language.

For a compiled language, a *compiler* translates the representation into an executable format, otherwise known as the *target program*. When executed by the computer processor (CPU), the target program accepts input, processes it according to the commands internal to the program, and finally produces the desired output. For an interpreted language, an *interpreter* executes the representation immediately. The interpreter simultaneously accepts both the source program and the input for processing at the point of execution. Hybrid compiler-interpreters also exist, where the source code is compiled into an intermediate format that is later interpreted by a *virtual machine* at the point of execution.

Compilation allows ahead-of-time optimisation of the code by the compiler, so the produced target program executes efficiently. Interpreter-based approaches are often used by tools used to debug compiler-based languages, by allowing a developer to step through their code, one command at a time. Purely interpreter-based languages cannot compete with compiler-based languages with respect to execution speed of the final target program.

For clarity, *compile-time* shall be defined as the point in time at which compilation is performed. *Run-time* shall be defined as any time that a previously compiled program is being executed on a computer. This latter statement infers that the program is loaded into computer memory and is being executed by the computer processor or virtual machine. Run-time excludes periods where the compiled files are simply residing in physical storage, such as on a hard drive or compact disc. The distinction between compile-time and run-time is essential to correctly utilising the features of the object-oriented approach for specifying software, discussed in later sections.

Compiled files suitable for execution fall into a number of categories. Two major categories are *executables* and *libraries*. *Executables* are programs available for immediate execution. Examples can include programs run from the command prompt of a textual operating system. Executables can also include graphical-user-interface applications run by a user on a graphical operating system.

Libraries provide sets of functions implemented for use by executables. Libraries centralise these functions to maximise their reuse and provide orthogonal building blocks to a developer writing software; each library is typically suited to a specialised domain, for example network programming. The libraries may be either (a) compiled into an executable at compile time—statically linked—or (b) linked to at run time by the running executable—dynamically linked. In either case, the library headers or interfacing code must be available at compile time, allowing the compiler to check that functions available within the libraries can be called later at runtime. Examples of libraries include dynamic-link libraries (.dll) found on Microsoft Windows, and library-object files (.lo) and shared-object libraries (.so) found on Unix. A java class file (.class) might either be an executable or a library, depending on choices made by the developer in writing the related Java source code.

Many programming languages provide basic data types with which a programmer can work, such as character strings, floating-point numbers, integers, and so forth. Variables of each type may be declared and assigned value within the source code. Note carefully that the variables declared in the source code become instances of their related types at run time within the target program. The instances only exist at run time: they may be seen as textual characters within the source code at compile time, but they don't actually exist as values in memory until run time.

Languages are either *strongly typed* or *weakly typed*. A *strongly typed* language rigorously enforces its type system, possibly allowing a restricted set of conversions amongst certain types; each variable declared by the programmer in the source program must have a corresponding type defined. For example, a variable defined to be an integer cannot have operations applied to it suited to a character string—unless the programmer applies an explicit type conversion upon the variable. This approach reduces error by ensuring that the developer correctly applies the available operations to the particular data—a quality often referred to as *type-safety*. In a *weakly typed* language, each variable declared in the source code does not require the associated type to be specified, or alternatively, the type of each variable must be defined but the compiler performs no type enforcement. In either case, the strictures imposed by strongly typed language do not apply.

The manner in which the type system is enforced is inherent to the language's implementation, and in turn dictates the style in which a programmer works with the language to achieve their goals. Enforcement implies that type errors are generated if the related type checker encounters the wrong use of types by the developer in their source code. Languages are either *statically* typed checked or *dynamically* type checked. The term "static" or "statically" when used as an adjective or adverb infers that the qualities of the related noun or action are determined at compile time. Hence, static type checking involves a type check at compile time: any type errors related to the misuse of types by the programmer are raised during compilation. The approach removes this overhead from run time, improving execution speed and reducing the size of the target program. Static type checking also makes more information available to the compiler, allowing further performance optimisations in the target program.

The term "dynamic" or "dynamically" when used in conjunction with other terms infers that the qualities of those terms are determined at run time. Hence, dynamic type checking performs the type check at run time, during target program execution. Certain languages allow types to be defined by the developer. If these types are defined in a separate source program unavailable to another developer at compile time, then a dynamically type checked language will still allow that developer to use those types in their own program, even though the type definitions are unavailable to them at compile time. This approach is unavailable to a statically type checked language.

While dynamic typing provides benefits, it does not eliminate coupling amongst types. Additionally, because statically-typed languages are more constraining than dynamically-typed languages, successful implementation-independent specifications of software models constrain their designs to suit static typing, because such designs, by default, may be implemented in any dynamically-typed language. The designs specified in subsequent chapters are designed around these notions.

Lastly, each language has rules governing the style by which variables are assigned, passed as arguments into functions, or returned as results from functions. A language supporting *value semantics* makes copies of the instances of variables in those cases. Alternatively, a language supporting *reference semantics* passes a pointer or reference to the instance of a variable—effectively passing the memory location of where the instance resides—allowing modification of the original instance rather than of the copy.

A C++ example of value-semantics behaviour is as follows. Assigning the integer value stored in variable *x* to the integer value stored in variable *y* using value semantics:

```
int y = x;
```

Passing the value *x* to function *func* using value semantics:

```
void func(int x) {}
```

A C++ example of reference-semantics behaviour is as follows. Referencing the value stored in variable *x* from reference *y* using reference semantics:

```
int& y = x;  // Using C++ references
```

and:

```
int* y = &x;   // Using C++ pointers
```

Passing the value *x* to function *func* using reference semantics:

```
void func(int& x) {}  // Using C++ references
```

and:

```
void func(int* x) {}     // Using C++ pointers
```

Languages can support various mixtures of both approaches, such as:

- allowing the developer (a) to specify which types support value and/or reference semantics and (b) to specify the intention wherever a variable is assigned, passed as an argument, or returned as a function result (C++).

- strictly enforcing that fundamental data types—integers, floating-point numbers—only obey value semantics and that all other types obey reference semantics (Java).

The semantic style interacts with other features provided by the language in accomplishing various design goals. Reference semantics provide a means to construct run-time representations of complex systems through the creation of in-memory graphs of type instances. The misuse of this facility can produce problems for the unwary programmer. If we copy the value of a type instance that is part of a larger graph of type instances, then this copied value—or *shallow copy*—is effectively degenerate: The copied instance becomes devoid of meaning in reference to the original graph due to the uniqueness of the original instance's relationships within that graph. For copied instances to have any meaning, a *deep copy* of the entire graph of type instances is required. This feature must be kept in mind when working in languages that support mixtures of reference semantics and value semantics.

The reference graphs of instances are essential to the present study because they allow the construction of in-memory representations at run-time. The present work argues that compilers be removed from simulation-assembly environments. Instead, through graphical representation and manipulation within GUI applications, practitioners can assembly run-time representations of their problems from predetermined concept sets.

2.2.2 Object-Oriented Software Methodologies

Software development frequently becomes a battle to manage the complexity of software design. Over the past few decades, higher-level software abstractions have been proposed to help in this task. An example is the proliferation of programming styles afforded by programming languages. A given language may facilitate imperative or declarative programming, along with various subclasses including functional programming, structured programming, logic programming, and object-oriented programming.

Of interest to this study is object-oriented programming (OOP), whose ancestry descends from the Simula 67 language (Dahl and Nygaard, 1966). The object-oriented style can be applied in almost any language (Holub, 1992), however, object-oriented languages contain features that facilitate this style more easily. The OO paradigm allows programmers to (a) define classes and create objects from those classes, and (b) define an assortment of relationships amongst these classes to constrain the collaborations amongst the objects of those classes. A *class* is an implementation entity that extrinsically defines, in a programming language, the attributes (data) encapsulated by a concept and the behaviour (functions or methods) exhibited by that concept. Each class has a unique identity. An *object* is an instance of a class that models an occurrence of the related concept. Each object has state and identity, and behaves in the manner defined by the related class. The state of an object is determined by the value of its attributes and each object has a unique identity.

OOP allows the explicit use of *encapsulation*—a software quality hinted at as far back as Wilkes *et al.* (1951)—to separate the specification of a class's behaviour from its implementation. A class has an *interface*—methods or functions that promise how the class will behave—and an *implementation*—data that provides state and algorithms that manipulate that state to provide behaviour. Encapsulation provides modularity, so that the implementation may be changed while the interface remains the same. In this way, complex software systems can be initially specified by defining all the involved interfaces, the task of implementating the interfaces divided amongst many programmers. A programmer on that project can then statically embed calls to the other interfaces when implementing their own portion of the software system.

Fowler *et al.* (2000) defines the four major types of OO relationships as follows:

- *Inheritance (Generalisation)*

  Inheritance is a relationship between classes, and is taxonomic—inferring a class-subclass relationship between classes. Inheritance infers an "is-a" relationship between classes. Many programmers consider inheritance a form of code reuse, in that the derived class inherits the functionality of the base class. Another form of code reuse arises from polymorphic behaviour inferred to the derived class in this relationship: under an OO language supporting reference semantics, an object of the derived class can be passed to a function defined to accept objects of the base class, allowing modification of that same object. Code reuse effectively arises from the reuse of code previously defined to accept base class instances. For statically-typed languages, inheritance is a static relationship defined to exist between two classes at compile time. Polymorphism is a dynamic behaviour exhibited by objects at runtime.

- *Association*

  Association is the most general form of relationship between objects of classes. Being such, associations may or may not infer ownership, and/or may or may not involve multiple objects at either end of the relationship. Often the objects at the association ends perform a specific role. While the definition of an association is expressed statically—at compile time—associations occur amongst objects, and hence only exist dynamically—at runtime. Association also allows polymorphism, because an object of a derived class can take the place of a given base class defined to exist at an association end.

- *Aggregation*

  Aggregation infers a whole-part form of association: multiple objects of a particular class form parts of an object of another class. The part-of objects often have a lifetime independent of the container object's lifetime.

- *Composition*

  Composition is a stronger form of aggregation where the contained objects belong to the container object: the contained objects only exist within the lifetime of the container object, being destroyed upon the destruction of the container object.

The significant features of OO languages—polymorphism, encapsulation, association, distinction between classes and objects—have distinct outcomes for the compile-time and run-time behaviour of a program, and require attention during software design. For example, the interplay amongst reference semantics and taxonomic polymorphism provide support to many design patterns—to be discussed in subsection 2.2.4. The author demonstrates this in C++, at the same time showing polymorphic class behaviour by defining inheritance of one class from another class and assigning objects of the derived class to pointers of the base class.

```
#include <set.h>
#include <iostream>

// Write code defining the base class A
class A
{
};

// Write code defining derived class B inheriting from class A
class B : public A
{
};

int main(int argc, char *argv[])
{
  // Create some instances of B
  B a;
  B* b = new B();

  // Polymorphically target pointer c at the same object as pointer b
  A* c = b;

  // Display the unique addresses of objects in memory
  std::cout << "Address of object aliased by var a: " << &a << std::endl;
  std::cout << "Address of object targeted by pointer b: " << b << std::endl;
  std::cout << "Address of object targeted by pointer c: " << c << std::endl;

  // Retarget c to the object aliased by var a
  c = &a;

  // Display the address of object now pointed at by c
  std::cout << "Address of object now targeted by pointer c: " << c << std::endl;
  return 0;
}
```

Note how class B subclasses class A—there is no notion of subclassing from objects in C++. Compiling this program and running the resulting executable:

```
amethyst > ./main
Address of object aliased by var a: 0xbfffdfd8
Address of object targeted by pointer b: 0x5907ca0
Address of object targeted by pointer c: 0x5907ca0
Address of object now targeted by pointer c: 0xbfffdfd8
```

As seen here, the memory address of each unique object is distinct and independent of any textual aliases such as variable alias *a* in the above code. The addresses displayed here would most likely be different when the program is executed in another process, such as on a different computer. Similar programs can be written in Java, Objective-C, and C#. to demonstrate the same behaviour in those language. In Java there is no direct memory access to display object addresses; however, a similar unique ID can be obtained for any Java object with the java.lang.Object.hashcode() method.

The object-oriented programming style encompasses more than an implementation technique. Features of the object-oriented approach have been extended to analysis and design, to produce object-oriented software methodologies (Rumbaugh *et al.*, 1991; Booch, 1994)—that is, approaches for realising useful software through defined processes involving the object-oriented paradigm. Within these methodologies, the terms object-oriented analysis (OOA), object-oriented design (OOD), and object-oriented programming (OOP) correspond to the sequence of development stages of analysis, specification, and implementation. Many such methodologies exist with considerable overlap in their approaches, the key features being (a) observance of the stages of analysis, specification, and implementation and (b) incremental and iterative steps amongst these stages. These methods differ from each other in the way that OO constructs are used in the three stages.

Analysis identifies the requirements of a requested software system for its useful application to a problem domain. Object-oriented analysis (OOA) (Booch, 1994) performs this identification through the classification of behaviour, state, and identity of concepts, into distinct types. A type is defined in terms of its behaviour, yet may be implemented in many different ways, using the classes of an object-oriented programming language (D'Souza *et. al.*, 1999). A class provides an implementation (realisation) of a type, or alternatively a group of classes might provide the implementation of a type. This dualistic relationship—or type-class dichotomy—corresponds respectively to language-independent specification and language-dependent implementation (D'Souza, 1997).

The design stage (OOD) follows that of the analysis stage (OOA). OOD involves correctly decomposing the results of OOA into a realizable design. In this case, the concept types from analysis are reconsidered from a logical/physical and static/dynamic perspective, for expression in an implementation-independent notation (Booch, 1994). Logical considerations involve identifying relationships amongst types and instances that satisfy the static and dynamic software requirements identified during analysis. Physical considerations involve identifying the concrete software and hardware components required in any implementation of the design. Like OOA, OOD is involved with types rather than classes—that is, specification rather than implementation. OOD selects one of the many permutations of type collaborations to provide a realizable design that meets the requirements generated during analysis. Because OOD involves types, some blurring exists as to where object-oriented analysis ends and object-oriented design begins, the two stages often being lumped into one stage termed object-oriented analysis and design (OOAD).

The implementation stage (OOP) follows the design stage (OOD) to implement the specified design as software. During OOP, the relevant physical components of the system are implemented as classes in programming languages. Additionally, relevant algorithms are selected and written to implement the internal logic of these classes—the remaining degree of freedom to be tied down in the developed software. The implementation stage is open ended: implementation infers continued maintenance, redesign, documentation, testing, debugging, and software release cycles to support the use and extendibility of that software.

The relevance of OO software methodologies to this study is their explicit use in the presented specification and implementation of the proposed infrastructure. Most importantly, this study recognises that the OOD stage is essential to this process, an issue that is further clarified in section 2.2.3.

2.2.3 Domain Ontologies and the Artificial Intelligence Community

The efforts of the software-engineering community have traditionally been paralleled by the work of the artificial-intelligence (AI) community. The last four decades have seen these communities exchange useful ideas regarding software expression and knowledge expression. This relationship has not always been collaborative: for many years, AI protagonists proposed that their own field performed knowledge engineering rather than software engineering (Musen, 2001). However, software must be applied or created at some stage to obtain value from knowledge extracted through knowledge engineering. By working concurrently to the software-engineering community, the AI community produced approaches and technologies that mirrored portions of software-engineering methodologies. These approaches include domain ontologies (Musen, 2001), knowledge-based systems (Lenat *at al.*, 1990), and expert systems (Heyes-Roth *et al.*, 1983), defined as follows:

- Domain Ontology: An expression of the high-level concepts, and the relationships amongst those concepts, for a particular discipline. Note the similarity to the classification of types during OOA. By "high-level" it is meant that logic, functionality, and detailed constraints are excluded from this representation.

- Knowledge-Based System (KBS): A software system designed for the expression and storage of knowledge bases. A knowledge base represents a collection of instances of knowledge expressed according to the concepts and relationships enforced by a domain ontology. From an OOAD perspective, this relationship mirrors the creation of concept instances from concept types. However, KBS enforces its pre-designed software on the user, replacing a portion of the implementation stage of software methodologies.

- Expert System: A software system that allows an expert to define logic regarding decision-making processes that draw upon user-supplied facts—as well as the content of an existing knowledge base—to automate the means by which intelligent decisions are obtained. This programmed logic provides the functionality excluded from the domain ontology, equivalent to identifying the relevant algorithms during the software implementation when shifting from OOD to OOP.

Because the AI community and the software engineering community have continued down concurrent paths, both fields have produced near-overlapping approaches that differ in their terminology. Additionally, by having already-established implementation tools, AI practitioners become overly comfortable with shifting directly from abstraction to implementation, without ever experiencing the design stage involved in software-engineering methodologies. Because the process-simulation community has adopted many ideas from the AI community, the above issues obscure the creation of simulation-assembly software when the necessary design practises of software engineering are applied to implement this category of software. Symptoms of this problem are exemplified by studies that declare themselves as both "knowledge based" and "object oriented", or which produce yet another modelling language and related compiler without considering alternative designs—see section 2.4.2 and 2.4.3 for further commentary. The present study distinguishes this potential problem during its own application of OO software-engineering methodologies.

On another note, the AI systems of the 1980's failed at ensuring the maintainability over time of knowledge expressed in those systems and did not scale well (Musen, 2001). As discussed in Chapter 1, software maintenance—the continued testing, refactoring, debugging, and documentation of source code, practised by software engineers—presents a critical task for the long-term reliability and extendibility of software. Due to the AI perspective familiar to the process-simulation community, practitioners of this discipline may mistakenly think that once-off prototypes of code, centralised within a knowledge-based repository, have immediate usefulness when integrated into future, unforseen, larger software systems. As will be elaborated upon in section 2.4.5, this approach ignores the importance of software maintenance and mistakes the real usefulness of repositories: repositories provide a means to shift coding and model implementation out of the practitioner's workspace and into the hands of remote software engineers. In this manner, the related software components can then be implemented, and more importantly maintained, in commodity languages for which a larger developer base exists. This argument represents one of the significant tenets upheld throughout the present study.

2.2.4 Additional High-Level Software Technologies

For completeness, three additional software terms shall be defined, due to their appearance in simulation-related literature: design patterns, component technologies, and agent technologies. The mention of object-oriented design leads to a discussion of *design patterns* (Gamma *et al.*, 1995; Buschmann *et al.*, 1996). Often the relationships amongst types refined during OOD produce recurring patterns of collaboration. A specific software design often involves an interaction of several such patterns to produce the desired behaviour. A developer perceptive of these patterns can more easily divide problems into constituent patterns, accelerating development and giving a higher chance that the appropriate design will be chosen on the first attempt. The formal categorisation of these recurring patterns aids communication amongst developers, by conveying complex software designs in concise, descriptive forms—in terms of their constituent patterns. Design patterns are a technique intended for the developer and often target the need to achieve specific runtime behaviour in a program.

Components (Szyperski, 1997; Heineman *et al.*, 2001) provide another software technology that gained popularity during the 90's. Many definitions exist regarding what constitutes a software component (Page-Jones, 2000), but generally, components are much like objects in that they are runtime instances of software entities. Like an object, a component exposes a well-defined interface—a set of public methods for access to the component's behaviour—for use with other components. Unlike objects, components possess a more clearly defined physical location. Additionally, components are deployed within containers that support component reuse. This support assists in the assembly, disassembly, and reassembly of components to collaborate in larger software systems, constructed at run-time. Significant container technologies enable the online discovery, access, and use by customers—CORBA (Object Management Group, 1998), J2EE (Sun, 2003), and .NET (Platt, 2003). The major push behind component technologies is to support the emergence of an online, web-services economy.

As a distributed technology, software components need not be downloaded for local execution, but rather executed at their respective physical locations. The container technologies provide a bridge amongst implementations written in different languages, as well as services necessary for handling recurrent problems that arise in distributed applications. This arrangement allows the implementation language of the component to be hidden by the component's public interface. In other words, a component need only be compiled on the platform intended for serving, allowing the use of a variety of implementation languages.

Agent Technology (Jennings *et. al.*, 1998) is a catch-all term for software components that assist a user to accomplish tasks by autonomously acting on behalf of that user. The number of applications of this technology is broad and remains open-ended. Being software components, agents perform their tasks at runtime, brokering with other agent-like components that act on behalf of other users or companies. *Mobile agents* (White, 1997) take this process a step further by allowing agent components to move through a network travelling from server to server in its quest to fulfil these tasks. The owner of the agent must define the objectives of the agent before sending them on their errand.

A major use of agent technology is to augment component technology: for example, agents tailored to resource discovery can locate and negotiate the use of online web services that provide component technologies. This approach furthers the possibility for a component-based economy to emerge. Agent technologies may achieve other goals depending on the needs of the particular domain. As will be seen in subsequent sections, agents have been applied in a specific manner to the field of simulation.

Regardless of their use, like other technologies in this section, agent technologies are not a single means to solving problems: agent technologies build upon existing infrastructure, and like many other approaches, their appearance results from evolutions rather than revolutions in computer science. Agents do not exist in a vacuum, as they must interact with existing technologies or infrastructures—such as components and component services—to achieve their goals. This communication must be expressed in concrete terms: an agent cannot deduce the intention of a component without a definition of that component's relation to a known concept set. It follows that while agent technologies are part of the future of the software community, they must still be implemented against concrete details. Hence, for component technologies and agent technologies, domain ontologies remain a hot area of research, as they provide a key ingredient to success.

2.3 General Simulation Software

Narrowing our view to the realm of numerical simulation, many of the aforementioned technologies have been applied similarly to manage the complexity implicit to assembling simulations. The most basic approaches involve programming languages within which practitioners write source code to express, and subsequently solve, simulation problems.

2.3.1 Modelling Languages

A variety of coding languages have been designed for modelling and simulation. Many of these languages fall into the category of *modelling languages*, a subset of programming languages that often—but not always—provide a declarative style of programming for an equation-based expression of problems. While often including the features of other mainstream languages, modelling languages usually provide a subset of these features, both limiting the programming styles available to the practitioner, and restricting the range of problem types that can be expressed in the language. This arrangement reduces error on the part of the practitioner when expressing simulations as code.

Often the related parsers, compilers, and interpreters are implemented in well-standardised programming languages—such as C, C++, and Java. The manner of implementation matches that of the approaches described earlier in this chapter: expressions within practitioner-generated source code are parsed into abstract syntax trees and subsequently transformed into a form amenable to numerical computation—such as matrices and vectors that can be solved using linear algebra.

Certain modelling languages address general simulation, rather than any one specific problem domain, incorporating an array of numerical techniques for solving problems from any discipline. Examples include Fortran (Backus *et al.* 1967) and Matlab (Mathworks, 1992). Other modelling languages are tailored to the needs of a particular discipline, simulation type, or mathematical model type. Examples include Modelica (Mattsson *et al.*, 1997) for process engineering, NSL (Weitzenfeld *et al.* 1999) for neural network simulation, as well as Modsim (CACI, 1997) and Simula (Dahl *et al*., 1967) for discrete-event simulation. These languages are bundled with additional libraries to solve the problems expressed in those languages.

Many of these languages are not implemented as command-line tools—that is, software executed from the command line or terminal prompt and that does not display a rich graphical user interface—but rather integrated as part of an application to control the practitioner's workflow. These *simulation environments* provide a blend of software development tools, AI-related features, and numerical solution techniques into the one software package. These applications resemble integrated development environments (IDEs) found elsewhere in the software industry, by grouping together a textual editor, compiler/interpreter, and debugger.

Alternatively, many libraries have been written in existing standardised languages, obviating the need for practitioners to learn a multitude of modelling languages and related software tools. Examples include Sim++ (Cubert *et al.* 1995), SimJava (McNab *et al.* 1996), and OOPM/MOOSE (Cubert *et al.* 1998), to name a few. Like modelling language, these libraries target specific problem domains or simulation types: Sim++ is a library applicable for expressing and solving discrete-event simulations in C++.

2.3.2 Component-Based, Agent-Based, and Distributed Simulation

The simulation community has also recognised the importance of component-oriented and agent-oriented technologies for improving computational performance through *distributed simulation*. Distributed simulation achieves higher computational performance through parallel execution—the distribution of a problem over multiple processors to collectively provide high-performance computing. This domain is also termed parallel computation and includes grid-based computing, an approach that extracts high-performance parallel computation through the use of wide-area, heterogenous, computing networks that share decentralised computing resources across organisational boundaries.

Many technologies provide services that facilitate parallel computation, both through commodity component technologies—CORBA (Object Management Group, 1998), COM (Eddon and Eddon, 1998), ".NET" (Platt, 2003), Java J2EE (Sun, 2003) and Java RMI (Sun, 2003)—as well as frameworks specialised to scientific computation that use their own custom inter-process communication protocols. These approaches include CCA (Allan *et al.*, 2002), PARDIS (Keahey and Gannon, 1997), and other "grid-based" frameworks (Foster and Kesselman, 1998). The goals of these various technologies are to (a) provide interoperability amongst computational libraries written in different languages, (b) coordinate parallel computational hardware—clustering—and/or (c) to facilitate grid-based computation.

The above uses of components address the solution of simulations rather than their assembly, and are not of interest to this study. It should be said, however, that such services are of importance for handing the responsibility of solving assembled simulations over to third parties, such as grid-based computational resources. Simulation assembly infrastructures—such as the one described in this study—would benefit from using such technologies to address issues related to solution.

Agent-based simulation has also received interest, particularly in the management of distributed-computing resources. Agent approaches—and related multi-agent approaches—have been proposed for (a) coordinating distributed simulation solution, (b) locating reusable models and domain knowledge for use in new simulations, and (c) guiding decision making in the assembly of simulations. These last two uses relate directly to simulation assembly.

On a side note, the architecture developed in this study has been termed an infrastructure for a purpose: the architecture underpins the sharing of resources amongst parties, choreographed by predefined protocols—that is, domain ontologies. As such, the infrastructure itself provides a foundation over which agent technologies could be layered. The possible addition of an agent layer presents an opportunity for future research.

2.4 Simulation Software for Process Engineering

As a subset of the general simulation community, computer aided process-engineering (CAPE) mirrors many of the technology trends from other simulation domains, which adopt technology from information technology and computer science. As such, the software is itself constrained by the same issues outlined so far in this chapter.

The section initially explores traditional approaches to CAPE simulation, leading subsequently to more recent developments in this field. Prior to this discussion, the work clarifies the two levels of granularity classically taught within process engineering for conceptualising problems from that domain. This differentiation has consequences of the ontology of this domain, and hence, has impacted upon the design of software written for this domain, and upon the manner in which software for this domain has evolved.

2.4.1 Process-Engineering Concept Granularity

It is intrinsic to the software tailored to process simulations that the ideas encapsulated by this software map to the domain ontology of process engineering. It is thus informative to first briefly consider the classically taught concept set applied to problems that arise in process engineering—the building-block concepts from which conceptual representations of process-engineering problems are built.

Two levels of concept granularity have classically existed within process engineering (PE). On one hand, the notion of the *unit operation* is often used to assemble representations of process plants. A unit operation is the traditional term for a basic processing unit typically found in a chemical plant. Examples of unit-operation types include the granulator, distillation column, filtration unit, pump, and chemical reactor. A process is often considered in terms of the unit operations constituting that process, along with material streams that flow amongst those unit operations. Unit operations provide a convenient unit of discretisation for an overall perspective of processes.

Alternatively, the interdisciplinary notion of the *system* provides a finer level of granularity in abstraction, and is often used to break down and analyse the internals of unit operations—although it is by no means restricted to this use. A shift to a systems viewpoint implies a stronger interest in the underlying mechanisms of a problem, rather than high-level abstraction. Many other fine-grained process-engineering concept types build upon and are encapsulated the notion of the system—particulates, thermodynamics, heat transfer, mass transfer, location of chemical reaction, and so forth.

2.4.2 Traditional CAPE Software

Over the years, the intersection of these notions with software technologies has produced the distinct software associated with process simulation. This software has traditionally combined three technologies for expressing and solving simulations: flow sheeting, modelling languages, and knowledge-based systems (Jensen *et al.*, 1998). Most traditional CAPE software is positioned within the spectrum of these technologies.

Flow sheet environments (FSE) depict a two-dimensional graphical representation of simulation problems within a running application—HYSYS (HyproTech, 2003), ASPEN PLUS (AspenTech, 2003), PROSIM (ProSim, 2003). This depiction is effectively a directed graph of nodes and connecting edges: the nodes represent instances of unit-operation types and the directed edges represent material streams that flow between these instances. The practitioner assembles their simulation by positioning unit operation symbols on the screen, corresponding to instances of the related unit operation types in their particular problem. The practitioner connects these instances with directed edges for the related material streams. These activities are runtime activities because they occur within a running application. The FSE maps the instances of the unit operations to mathematical models that are either (a) written by the implementers of the FSE and compiled into the tool, or (b) expressed by the practitioner in a modelling language in adjunct to the tool—explained shortly.

One could consider flow-sheet environments a visual language depicting runtime instances of a restricted set of concept types. Why restricted? Traditionally, FSE tools had the set of unit-operation types compiled into the tool, restricting the FSE to this particular level of granularity. Where a modelling language was provided, such languages were only intended for expressing equations. Any lower level of granularity was intrinsic to the mathematical equations, but not explicitly available for use in the flow sheet: the practitioner could not graphically manipulate instances of systems on the screen, but rather implicitly code their qualities into the equations. The same applied to low-level concepts that build upon the notion of systems—such as spatial variations, heat and mass transfer mechanisms, and so forth.

As explained earlier, modelling languages (ML) were often intrinsically tied to more advanced FSEs—gPROMS (Process Systems Enterprise Ltd., 1999)—or alternatively provided as standalone modelling IDEs to assist in the coding, compilation, and debugging of simulations—ASCEND (Piela, 1989), MODEL.LA (Stephanopoulos *et al.*, 1990). It was rare for MLs to be implemented as an independent library for inclusion into different environments. This arrangement made many MLs specific to the simulation environment with which it was distributed, leading to reduced interoperability amongst tools.

Many MLs for process engineering were tailored for expressing and solving a mixed set of both algebraic equations and ordinary differential equations (ODEs)— DAESIM (Daesim Technologies, 2003). These types of equations are common to the mathematical models of the finer-grained concepts described earlier.

Because purely FSE approaches were traditionally locked to a particular set of abstractions, knowledge-based system (KBS) approaches subsequently aimed to provide an extendibility of knowledge within the system. This misapplication of the term "knowledge base"—see section 2.2.3—has led since to the confusion most readers make between OOP and KBS approaches within this field. Subsequently, because the definition of concepts inherently required the specification of mathematical models for each concept type, these languages blended OOP with the features of modelling languages—DESIGN-KIT (Stephanopoulos *et al.*, 1987), OMOLA (Andersson. M., 1990), DYMOLA (Elmqvist, 1993), Modelica (Mattsson *et al.*, 1997). The implementation of KBS approaches as an OOP/ML styled language is often provided as an interpreter or compiler built into the related simulation environment.

Such approaches also include code transformation that parses an OOP-styled conceptual representation of a problem to generate equation-based ML code, which is then processed by the parser or compiler for that ML—an example being Schema (Williams *et al.*, 2001).

Regardless, this route produces a number of common side effects. First, often the definition of the concept types is statically bound to the mathematical-model type in the OOP/ML code: a change to the mathematical model type for a given concept type involves changing code for that concept type. This approach runs counter to defining the concept type in one place—in terms of its assembly rules in relation to other concept types—then implementing a multitude of different mathematical model types separately for that concept type. This latter approach makes better sense from a knowledge management perspective: the definition of a concept set in one place cuts down on duplicate code in implementation, improving maintainability by reducing the number of places within the source code that require change when the behaviour for a particular concept type is changed.

Second, by providing an OO-styled programming language in the runtime simulation-assembly environment, the separation between compile-time and run-time becomes indistinct: where concept types were previously compiled into the environment—as in traditional flow sheet environments—the environment now facilitates any number of recompilations of concept-type definitions. Subsequently, matters regarding OOD become more difficult to analyse, in particular which OOD heuristics should be applied. These heuristics are essential to the correct application of OO language constructs, specifically the trade off amongst inheritance, polymorphism, association, and composition described earlier in this chapter.

Third, providing this level of extensibility in a single runtime environment places choices for (a) concept-type definition, (b) mathematical-model definition for each concept type, and (c) assembly of the specific simulation from instances of these concept types, all in the hands of the practitioner. The former two tasks are similar to writing libraries of types for others to use, while the latter involves creating and assembling instances of those types. Software developers understand the necessity to clearly delineate the tasks of the library creation from that of library use, to maximise reuse of library code.

Delineating the client developer from the library developer helps to rationalise the design of an appropriate type system for a domain. Rarely does a practitioner understand these ramifications, leading to a poor partitioning of the goals of (a) defining the concept types for a discipline prior to compilation from (b) specialising the attributes of runtime instances of those concept types to describe a specific problem—an effect known as the type-instance dichotomy. There is also little preventing the practitioner from lumping together concept types that exist on different planes of granularity—for example, unit operations with systems—without attention to correctly restricting their interactions.

The acceptance of this level of extensibility impacts upon the practitioner: the assembly of process simulations remains a complex and error-prone task. The present study argues that allowing endless extensibility within the simulation assembly environment causes more problems than it solves. The endemic acceptance of this approach is exemplified by Jensen *et al.* (1998) who asserts that model construction involves the following steps: (1) "Decompose the model into building blocks", (2) "Create new building blocks if they do not already exist", and (3) "Aggregate the building blocks into the model". Note that little differentiation is made here between what constitutes a type of building block as opposed to what constitutes an instance of a building block—a lack of awareness of the type-instance dichotomy. Acceptance of the above three steps impacts upon the complexity of the simulation-assembly environment: it condones the act of practitioners defining knowledge regarding their discipline, as opposed to using this knowledge for assembling descriptions of individual problems. It mistakenly condones the inclusion of knowledge-definition tools within the simulation assembly environment, as if this were a fait accompli.

In an abstract sense, practitioners draw from a known set of concept types from their discipline. The act of breaking problems into so-called "building blocks" should be restated: practitioners break problems into the *known* set of concept types that they draw from to build their simulation. Within the context of abstract knowledge in a particular field—that is, away from issues of software implementation—it is rare for a practitioner to create completely new concept types: many fields are already characterised, with the abstract concept set standardised through consensus, as reached by the leading thinkers in the respective fields. The definition of a specific simulation problem is orthogonal to this task, as it draws upon these standardised concept types, effectively using instances of those types.

It could be argued that knowledge-based simulation environments can be tailored to any domain, but if we are providing a simulation environment for process engineering, is this former idea our real goal? Considered in this context, the effort to provide extensibility within the simulation environment appears misplaced. It is really the act of getting standardised information—regarding the specific domain ontology—implemented correctly in the software or read into that software. As will be seen shortly, attempts in the literature have been made to characterise the ontology of process engineering.

2.4.3 Process-Engineering Ontology

With the exception of the standardisation efforts discussed in the next section, literature regarding process-engineering ontology has classically appeared as isolated studies. Several works have proposed taxonomic hierarchies of concept types to characterise the relationships amongst many process-engineering concepts—such as VEDA (VEDA Team, 1999) and CIPOS (Qian *et al.*, 2000). Many studies inadvertently approach this task from an AI perspective, rather than from a software-engineering perspective. Section 2.3.2 has already emphasised the problem this creates. Hence, these studies have been performed from an OOA viewpoint, meaning that without revision of the related hierarchies from an OOD viewpoint, the documented ontologies present an initial choice requiring further decomposition to be subsequently implemented. The symptoms of this OOA perspective reveal themselves in the related type hierarchies, which contain breaches of various OO design heuristics outlined by Riel (1996), such as:

- deep inheritance trees involving every type in the related framework deriving from a central "god" class.

- the presence of "agent-like" types that would be eliminated from these hierarchies in applying iterative software development.

- Unnecessary cyclic relationships amongst types.

Various types and relationships produced in these hierarchies would be found to be irrelevant at design time, a feature of most analysis-generated models. While an informed reader may insist that software implementations have been produced from these works—ModKit (Bogusch et al., 2001)—it must be highlighted that these works have acquiesced from the OOD process by reproducing the static/dynamic and physical/logical design split of existing process simulation software. These features frequently involve the production of a single-vendor compiler for a niche programming language placed in an IDE that is collectively incompatible with other simulation environments without the creation of bridging libraries.

This situation perpetuates the standalone application approach, a situation that would change if thought were applied at design time, to centre the tool around commodity software technologies and to remove of the practitioner from the role of coder—as will be elaborated upon in Chapter 3.

2.4.4 Recent Technology Developments

The appearance of component technology, and related agent technology, has likewise been reflected within the discipline of computer-aided process engineering. Of prominence over the last few years has been the specification of CAPE-OPEN standards (CO-LaN, 2003). CAPE-OPEN specifies a component-oriented approach to reusable software for process simulation. In this case, key concepts related to process engineering and process simulation have been characterised with a standardised set of component interfaces. This standardisation provides interoperability by allowing runtime integration of these resources into software, provided that the related software conforms to these interfaces. As mentioned earlier, an "interface" provides a concrete definition of the behaviour of a component, hiding the implementation of that component.

Being an object-oriented technology, components face the same design issues that arise in OOP: as a runtime entity, a component's functionality is statically defined by an interface to exhibit certain runtime behaviour. This is a key issue because the ontology used in CAPE-OPEN is statically locked in, according to the specified standard—otherwise, interoperability would not be possible. As such, CAPE-OPEN must define a lowest level of concept-type granularity with which to build problems in running environments that the use related components. In the case of CAPE-OPEN, this building -block type is the same as that of the original flow sheet environments—namely, the unit operation. It follows that a bare-minimum simulation environment, constructed solely from third-party components that implement the CAPE-OPEN specification, would provide the same modelling granularity as a traditional flow-sheet environment.

The incentive promoted for the specification of CAPE-OPEN is the reuse of both legacy code and legacy databases of physical property data. Like other simulation domains, much process simulation software has already been implemented as libraries in older languages such as Fortran. Using class wrappers from contemporary OO languages—for which CORBA (Object Management Group, 1998) and DCOM (Eddon *et al.*, 1998) bindings exist—older code may be made interoperable with more recent applications. It is also envisioned that more recent simulation environments will adopt this standard to improve interoperability amongst existing simulation-assembly environments, both by using third-party components, and by allowing models implemented within these environments to be exposed as CAPE-OPEN compliant components to other tools. These latter uses aim to improve the interoperability amongst existing simulation environments, a long-standing deficiency in CAPE-related tools (Wedel *et al.*, 2000).

It must be stressed that standards such as CAPE-OPEN facilitate interoperability: they do not provide a means to implement the defined interfaces of the components—this is the job of third-party vendors. An interface only provides a public definition of behaviour to be expected in using a component that implements that interface. Hence, CAPE-OPEN does not replace the need to implement models of individual unit operations, particularly when designing new, novel, complex, or as yet unmodeled designs of the various unit operations—such as the converter model developed in this study. Additionally, by being based upon the granularity of the unit operation, CAPE-OPEN provides a black-box view to simulation. Like flow-sheet environments described earlier, the finer grained concepts such as systems have been purposely excluded, because a black-box perspective excludes model implementation of the respective unit-operation components is irrelevant to CAPE-OPEN.

As a standard, the CAPE-OPEN specification is also separate from the underlying component technology, which manages components that implement the defined interfaces. While the specification presently defines CORBA and DCOM interface definitions, it is likely that as J2EE (Sun, 2003) becomes more prominent—and because it is interoperable with CORBA—the CAPE-OPEN specification will most likely also be implemented as J2EE components—known as Enterprise Java Beans (EJBs).

Agent technologies have also been proposed for assisting the practitioner in the assembly process simulations. Because agents are software components, they require a defined ontology against which to work. Understandably, CAPE-OPEN provides one such basis, utilised by the ONTOCAPE and COGENTs projects (Braunschweig *et. al.*, 2002). It is proposed that runtime COGENTs agents will query and interact with components that obey the CAPE-OPEN specification. At the time of writing, both the COGENTs and ONTOCAPE standards are projects under research and not as yet published. Remember again that agents cannot operate without a statically defined agreement on the ontology of the related components with which they cooperate. The COGENTs project is thus inherently tied to the ONTOCAPE and CAPE-OPEN ontology. Hence, COGENTs is similarly limited to providing advice for building process models at the unit-operation level of concept granularity. Without explicit and static definitions of finer-grained concept types, COGENTs aid practitioners in assembling process models built from unit operations and equation-based expressions of finer-grained concept types.

2.5 Summary

This chapter has reviewed background knowledge and literature related to simulation software. The concurrent efforts of the artificial intelligence community and the software engineering community have created confusion for practitioners attempting to apply the related technologies. In particular, efforts to create simulation-assembly environments have been affected by insufficient attention to the stage of object-oriented design in the various software development methodologies. Arguments have been presented to propose that practitioners should not be involved in the act of coding and knowledge extension. The next chapter analyses and designs an infrastructure to address these problems.

Chapter 3: Analysis and Design

Chapter 2 raised questions regarding the effectiveness of coding languages to practitioners. The work also questioned the suitability of practitioners as candidates for performing knowledge extension within their simulation-assembly software. The present chapter recommends the removal of coding and knowledge extension from the practitioner's workflow, through the elimination of coding languages, compilers, and knowledge-extension facilities in the simulation-assembly environment. The work analyses and designs an infrastructure supporting this approach, culminating in the specification presented in the next chapter.

3.1 Important Observations

The discussion of the previous chapter highlights problems in allowing practitioners to perform coding when assembling simulations. The key issues raised relate to the following problems:

- The provision of coding in the assembly environment imposes design choices and development practices on the practitioner that are outside of their skill set.

- The provision of a compilation step in simulation-assembly environments blurs the distinction between run-time and compile-time, further affecting these design choices.

- Time spent on software development by practitioners detracts from the task of simulation assembly.

- The collective code for a simulation requires continued software maintenance to ensure its long-term value. Code must be understandable and relatively bug free for its modification, extension, and comprehension by both the originating practitioner and by other practitioners in their workplace. However, practitioners often inadvertently neglect these maintenance tasks following the initial creation of working code.

- The creation of niche modelling languages, and their related compilers, further fragments the simulation community, due to incompatibilities with existing modelling software.

- The creation of niche coding languages detracts from the maintainability of the related simulation code, due to an inherently smaller developer base and supporting toolset, compared to commodity programming languages. The latter toolsets assist in software development purposes such as testing, debugging, profiling, and documentation.

Furthermore, Chapter 2 has questioned the suitability of practitioners to perform knowledge extension within simulation-assembly tools. Exposing such a capability invites the misapplication of the object-oriented constructs implicit to these approaches. The problem is compounded by the blurring between compile-time and run-time, introduced by the mixture of flowsheeting and compiler facilities within many simulation-assembly environments. These problems also blur the distinction amongst the following tiers inherent to assembling simulations:

- Tier 1: The assembly and construction of simulations for the problem of interest—modelling.

- Tier 2: The definition of the set of concept types and mathematical-model types for use in the assembly of simulations—meta-modelling.

- Tier 3: Definition of the constraining architecture in which (a) these modelling and meta-modelling tasks can be performed and (b) solution algorithms are applied to assembled models in obtaining numerical results.

One solution to these problems is to avoid exposing the coding languages, compilers, and knowledge-extension facilities to the practitioner. This constraint contains the activities of practitioners to within the first of the above three tiers. Additionally, the removal of any compile-time step from the practitioner's workflow means that simulations must be assembled purely at run-time. As will be explained in section 3.5, run-time assembly involves the practitioner directly manipulating an in-memory data-model, where a graph of run-time instances of concept types is constructed to represent the simulation problem. The instances form a graph by holding run-time references to other instances in that graph. Most run-time software uses this technique in its implementation. The exclusion of any compile-time step requires that the assembly environment does not provide any interpreted or compiled language to construct this in-memory graph. This

arrangement necessitates that the practitioner use only a graphical-user interface (GUI) to perform this task, with no parsing of user-supplied expressions at any place in this GUI.

While the removal of a compile-time step eliminates coding on the part of the practitioner, this approach is insufficient to eliminate knowledge extension performed by the practitioner at run-time. For example, graphical approaches exist to express knowledge through visual depictions of the various OO constructs. To exclude these approaches, we must further stipulate that no means of run-time knowledge definition be exposed to the practitioner in the assembly environment. However, this stipulation is not intended to banish the implementation of related data models beneath the assembly environment. Such facilities are essential to loading in-memory representations of knowledge created by roles other than the practitioner. These facilities are essential to constraining the construction of the run-time graph of instances of the related concept types.

The removal of both concept-type definition facilities and the coding languages for implementing mathematical representations of concept types, implies a physical design constraint: the related compilers for such languages have been removed from the simulation-assembly environment. The removal of these physical software components infers that tasks related to their use are also removed from the duties of the practitioner. However, defined concept-types and technique-type implementations must somehow be accessible to the practitioner, for their instantiation and use to assemble simulations. The design of this accessibility must obey constraints present so far, most importantly that no compilation stage be involved in the practitioner's workflow. To achieve this requirement, we must consider these static and dynamic software properties in the context of the high-level workflows, classically applied by practitioners when assembling simulations.

3.2 The High-Level Workflow of Simulation Assembly

The modelling of reality combines many theoretical concepts to describe phenomena. Practitioners from different simulation disciplines develop sets of mental concepts to apply in modelling problems specific to their field. A practitioner selects an appropriate assembly of concepts to describe a given problem, and then applies mathematical techniques to represent those abstractions. Additional mathematical methods are then used to solve this representation. The approach is usually iterative: revisions to the choice in abstractions lead to changes in the mathematical representation. This activity continues until a suitable model has been obtained, achieving an adequate representation of reality with a minimum of model complexity.

As seen from this procedure, model assembly involves two types of tasks:

1. The layering of instances of concept types over real-world processes, to construct a conceptual representation of the problem.

2. The layering of instances of mathematical technique types over these concept instances, both to represent the concept instances and to solve the problem.

A multitude of conceptual descriptions may exist for a particular problem, based on the selection of concept types and the creation of concept instances. Additionally, a variety of mathematical representations may exist to represent a given conceptual description. In other words, variability exists at two levels. This arrangement introduces a dependency, whereby choices in the mathematical layer depend upon choices made in the conceptual layer—see Figure 3.1.

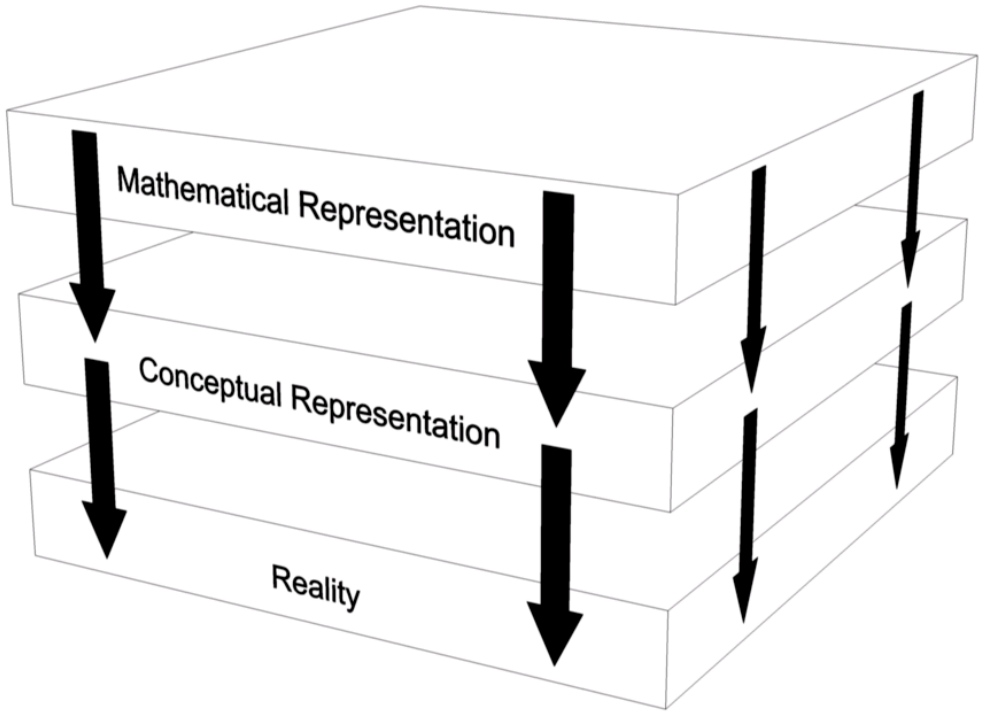


Figure 3.1: Depiction of the task-type layers involved in modelling. The arrows represent the direction of dependency in the related choices.

Model selection proceeds from a simple starting point, with complexity increased as the influential mechanisms in the problem are identified. A simulation's design thus moves through a spectrum of completeness. Any run-time simulation assembly environment should allow the practitioner to initially create simple representations of their problem, to which complexity can be added to obtain an adequate representation. Within this representation, different parts of the same simulation should be allowed to exist at different degrees of completeness, while not producing invalid assemblies.

3.3 Architectural Partitioning

Returning to our discussion of section 3.1, we can now propose a suitable high-level design for the architecture. First consider the two levels of variability in creating simulations, specifically (a) to create instances of concept types for assembly, and (b) to select, instantiate, and associate mathematical-technique types to represent instances of concept types. The simulation-assembly environment must accordingly allow the practitioner to perform these tasks. When combined with the requirement that no compilation step exist in the practitioner's workflow, the desired design must not statically couple the technique type with its related concept type. At run-time, the practitioner must be able to change mathematical technique types to represent individual instances of concept types. Hence, if the creators of the assembly environment compile the concept type and the technique type into the one class, a practitioner can only select a different technique type by recompiling the application's source code to contain their own mathematical implementation. This situation would violate the requirement that no compile-time step exist within the practitioner's workflow. This is a difference between the present work and approaches such as SCHEMA (Williams *et al.*, 2001).

The design must therefore statically decouple the mathematical-technique types from the concept type they represent, so that (a) instances of concept types can be created at run-time and (b) the choice, instantiation, and association of related technique types can also be performed at run-time. However, for the practitioner to create run-time instances of existing concept types and technique types, existing domain ontologies and technique types must already be defined and implemented.

To address this task, we must first divide the traditional responsibilities of the practitioner amongst three roles in accordance with related technical expertise—see section 3.4. Second, with the removal of modelling-languages and their compilers from the assembly environment, there no longer exists the need to provide a softened language to match the practitioner's related technical level. Instead, ontology definition and technique-type implementation can be performed externally in commodity languages.

While a suitable design might still compile these statically decoupled techniques and concept types into the assembly environment, this arrangement unfortunately provides a finite range of techniques to the practitioner. However, the range of mathematical models for various concept types is open ended: published literature grows with the discovery of new models of different concepts. This dilemma creates an important requirement: the architecture must allow a developer to systematically extend the assembly environment with new mathematical techniques, when these methods appear in published research. Combined with the restrictions regarding compile-time constraints discussed so far, this need requires the publication of technique-type implementations, to allow (a) access by practitioners on demand, and (b) access by developers implementing new techniques that depend on existing techniques. Note that the former point implies some form of systematic cataloguing to allow the practitioner to search and locate appropriate techniques.

Furthermore, with many disparately located developers implementing individual techniques, some means must be used to centralise the agreed concept set against which these techniques map. Practitioners must be able to assemble instances of individual technique types into larger software systems. Hence, developers must share an agreed protocol on the available concept types—the domain ontology—for coherent run-time assembly of implemented techniques.

Chapter 2 has already explained that a pre-defined set of concept types for a domain, along with the high-level relationships amongst those concept types, is termed a domain ontology. The use of this term in the present study further infers the exclusion of mathematical concepts—such as the terms "variable" and "equation"—which participate in the implemented algorithms to represent concept types, and which are coded by developers rather than expressed in ontologies. For example, when considering a domain ontology for process engineering, we discuss concept types such as "system", "stream", and "convection".

The next problem is to make these external resources accessible to the practitioner at run-time. The solution is to store domain-ontology definitions and technique-type implementations in external repositories, for remote access in the assembly environment. By further physically decoupling these repositories according to their resource type, the standardised domain ontologies may be centralised for third-party vendors to implement technique types against concept types within a given domain ontology. The techniques are deployed in their own repositories as executable components, for either download or remote execution by the practitioner. In this way, a specific domain ontology provides a high-level protocol against which third party can implement disparate techniques for integration into larger systems assembled by the practitioner.

Additionally, for practitioners to access to in their assembly environment at run-time—configuring their environment— to select the concept set and relationships displayed in the environment from which they can create instances and assembly into graphs of concept instances. This approach statically defines the protocol against which developers implement techniques that represent those concept types, enabling these representation to be implemented separately and to be assembled in the practitioner's simulation assembly environment at run-time.

The technique repositories provide more than once-off locations to publish implemented components for run-time integration. Having the ability to implement technique types in commodity languages raises both the available developer base and improves the chance for long-term maintenance by software engineers to increase reliability. Furthermore, by being served from their respective physical locations, the published techniques can provide bridging to high-performance computing facilities, such as those mentioned in section 2.3.2.

3.4 Role Revisions

By taking the tasks presently imposed on the traditional role of the practitioner, and dividing them amongst a new set of distinct roles, we can support the infrastructure proposed in the previous section. As explained in section 3.1, the practitioner performs tasks residing within the modelling tier, our goal being that they play no part in the meta-modelling tier. Meta-modelling in the context of mathematical simulation relates to (a) defining the concept types from the practitioner's field and (b) implementing mathematical model types that exist for each individual concept type. It follows that these two parts of the developer's role can be divided into two further categories:

Architect: The architect identifies the set of concepts particular to the practitioner's field. They use their knowledge of software engineering and OOP to clearly select a data structure mapping the concept types and relationships amongst these types. The architect carefully chooses these concepts to ensure an appropriate granularity of types, so that the practitioner can assemble any conceptual representation of their problem at run-time. The practitioner distributes this domain ontology of concepts to developers. The architect may be more than a single person: it may be a standards committee consisting of a centralised decision-making policy, with regard to the structure of the particular ontology.

Developer: A developer codes mathematical models for a concept type within a domain ontology. The client programmer rigorously tests these models before the model is declared suitable for distribution to practitioners. The models are selected by the practitioner at run time, to model the concepts selected by the practitioner for their assemble problem description. The idea parallels that of plugins and modules found in other applications.

The proposed relationships amongst architect, developer, and practitioner are depicted in Figures 3.2, 3.3, and 3.4 as use-case diagrams. This new arrangement removes the practitioner from coding. The practitioner instead uses a running application to perform run-time assembly of simulations. A key to the success of this arrangement involves developers correctly choosing a type-granularity that provides adequate run-time variability for practitioners, so that practitioners need never involve themselves in coding additional concepts or models to support their problem domain.

The structure lends itself to better reuse of implementation, because ontology identification is now (a) performed once, (b) more likely to be implemented correctly, and (c) more easily centralised for remote access by practitioners using running applications. This arrangement runs counter to those OO designs whose implementations allow practitioners to create a multitude of disparate domain-ontology equivalents, as practitioners are allowed to add concepts and models that are either (a) poorly factored, or (b) actually already supported but unknown to the practitioner.

The practitioner uses the architecture in the following manner. Within a running application, they choose the ontology of concept types suited to their problem domain. They then proceed to assemble a conceptual representation involving instances of these concept types. The practitioner performs no coding through use of a graphical-user interface. During this process, the practitioner also selects mathematical technique types to represent these instances of concept types. Developers are assumed to have implemented these techniques prior to the practitioner's efforts and independent of their problem. This assumption can be made due to the ubiquity of these techniques: the mathematical methods are not just applicable to the particular problem, but to many other problems that arise in that problem domain.

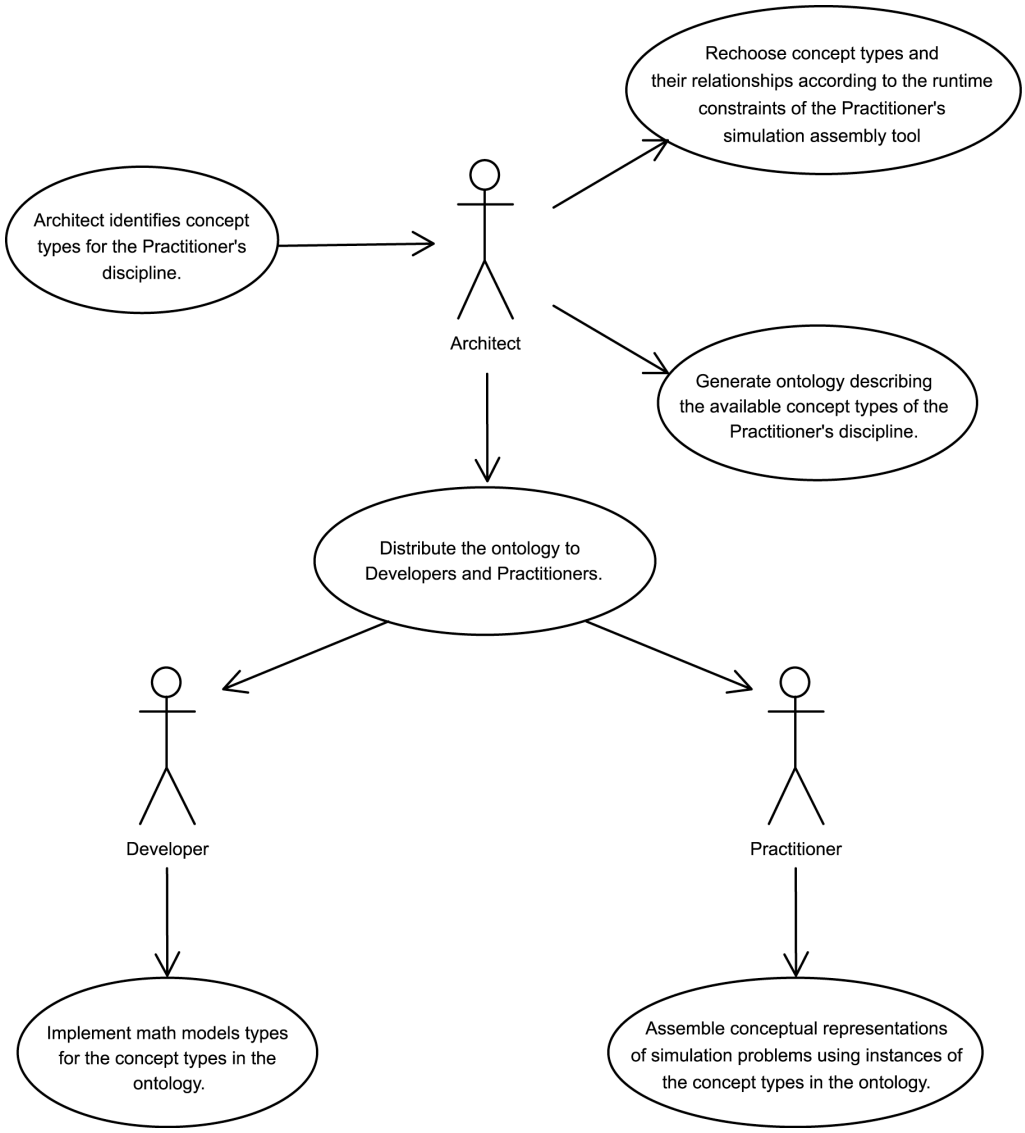


Figure 3.2: Use-case diagram depicting the role of the Architect.

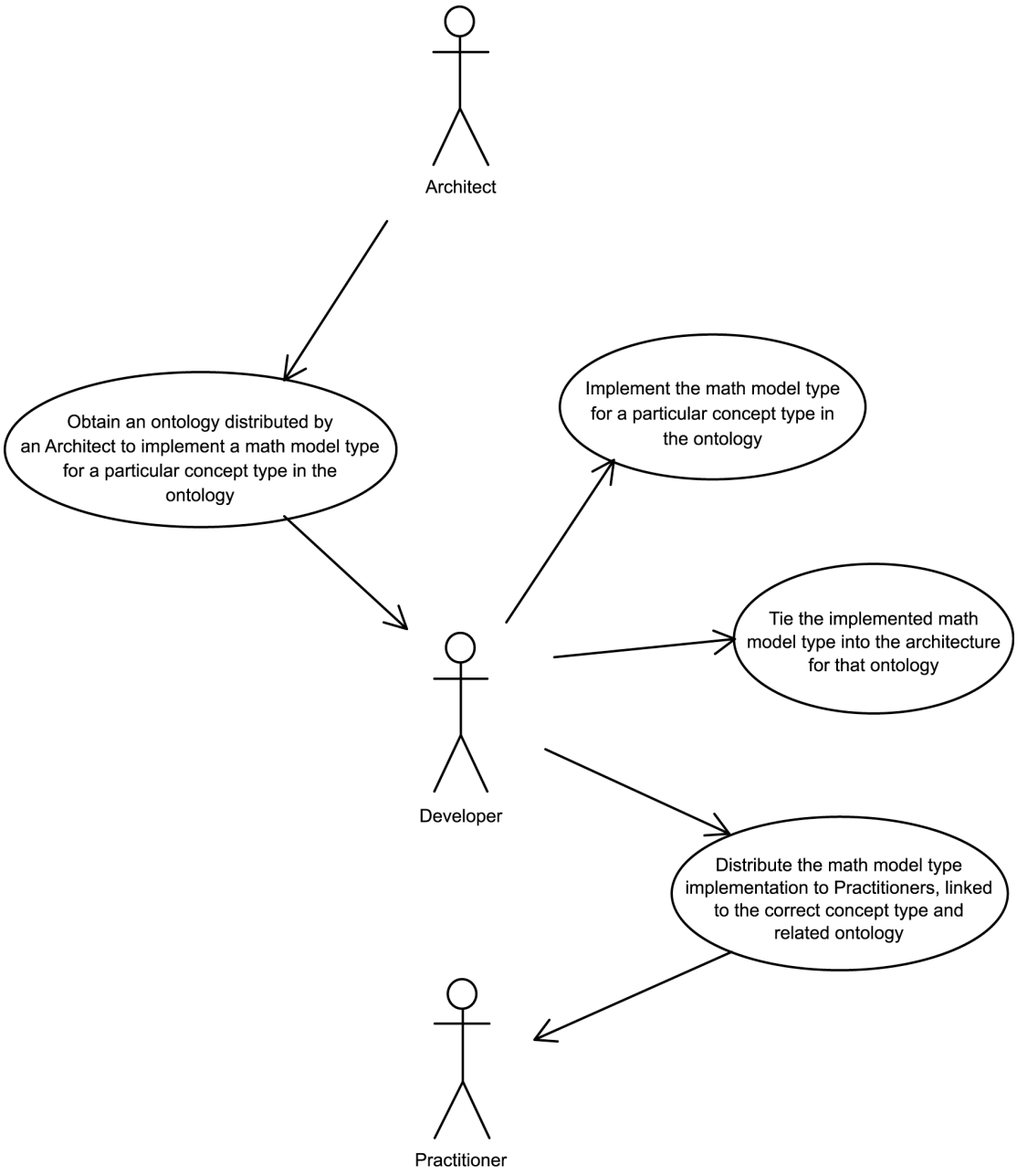


Figure 3.3: Use-case diagram depicting the role of the Developer.

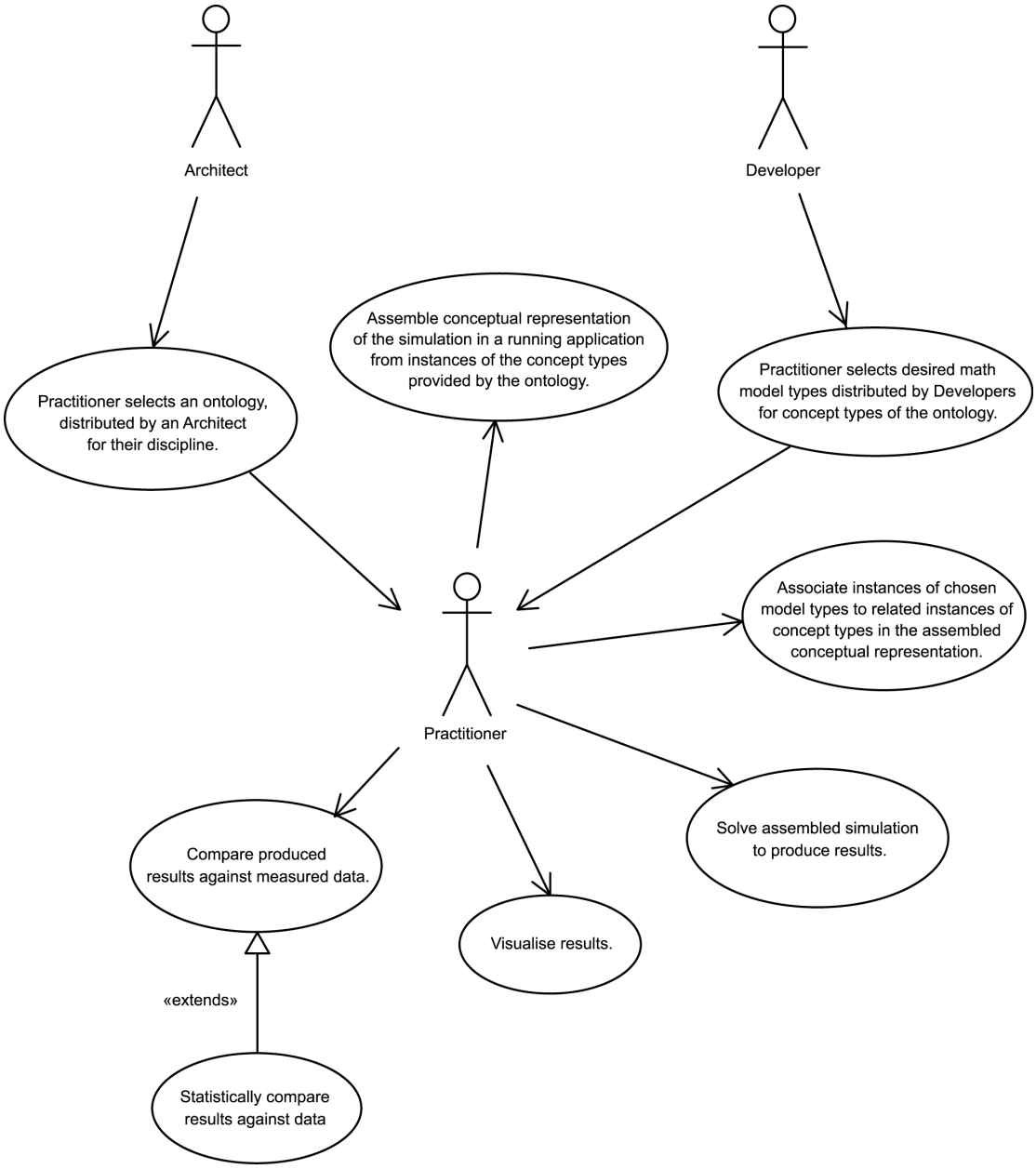


Figure 3.4: Use-case diagram depicting the role of the Practitioner.

The practitioner experiments with a variety of mathematical methods—implemented by developers—to experiment with different mathematical representations for each concept. We do not explain the nature by which these mathematical techniques are obtained: the answer is implementation dependent and not addressed by this document. However, the method of repositories outlined earlier highlights one approach, with domain ontologies and mathematical methods stored in remote, searchable repositories, or alternatively as individual services remotely accessed from various developer-owned servers. In this manner, the practitioner could have access to a broad range of techniques that are abreast of the latest findings related to their particular phenomena.

3.5 Visualisation and Data-Model for the Assembly Environment

With the high-level infrastructure identified, the study next considers design of the simulation-assembly environment, hereafter simply referred to as the assembly environment. The goal is to present run-time visualisation of simulation assembly to the practitioner, according to the rules of any domain ontology loaded into the assembly environment. As already established, the assembly environment must provide a GUI presentation of simulation assembly, to satisfy the run-time requirements of section 3.1. The assembly environment accepts pre-defined domain ontologies supplied by architects, to constrain the gamut of concept types from which practitioners create run-time instances. The design must constrain (a) the data model that architects use to define the ontologies which (b) influences the data model within which practitioners assembly run-time conceptual representations. These goals ensure that the assembly environment provides an intuitive run-time GUI visualisation for practitioners to (a) assemble the in-memory graph of concept instances, (b) select, instantiate, and associate technique types to represent those concept instances, and (c) construct these assemblies into valid structures.

The object-oriented constructs available to the design include *composition*, *inheritance, parameterisation, association*, and *polymorphism.* The complimentary use of these constructs leads to different run-time and compile-time outcomes. Our goal is to identify the correct balance of these constructs that together allow adequate run-time flexibility for the practitioner. The first three constructs must be excluded because of their inherent coupling to a compilation step. Composition, inheritance, and parameterisation involve static binding within a coding language, automatically breaking the restriction on compilation by the practitioner. Although certain graphical, run-time environments do allow problem specification using these constructs, this exposes facilities permitting knowledge-extension by the practitioner, open to ineffective use and confusion, and breaching the constraints presented in section 3.1.

In comparison, the association construct does allow run-time assembly. Under this approach, the practitioner creates instances of types at run-time that hold references to other instances created previously by the practitioner. The types in this case are concept types and technique types; the instances are unique occurrences of the concept types and technique types within a representation of a specific simulation problem. The resulting graph of associations amongst instances represents an expression of the overall simulation problem. When combined with polymorphism—utilised by the architect in defining the domain ontology—a practitioner can be provided with a sufficiently appropriate run-time environment to assemble simulations. Note that, although polymorphism is a run-time consequence of inheritance, the use of this facility is the responsibility of the architect within their static definition of the domain ontology. Hence, in terms of practitioner responsibilities, no violation of the run-time-assembly requirement occurs.

The visual graph of instances would escalate into an unworkable clutter for the practitioner, if not for some form of order. This problem becomes pronounced when practitioners must construct their graph of instances based on ontologies containing many different concept types. The architecture should provide hierarchical, formulaic assembly of instances, rather than an unstructured collection of abstractions. Ordering should ensure a minimal graph of associations, which contain no duplicate information to minimise the probability for error by the practitioner.

The solution proposed by the present study combines two classic techniques for decomposing systems, specifically layers (Dijkstra, 1968) and Bayesian networks (Jensen, 1996). Layering assists in the decomposition of systems into manageable parts and creates reusable modules (Dijkstra, 1968). Alternatively, Bayesian networks consist of nodes connected by directed, acyclic edges, which can represent directed dependencies and relationships. When applied to the design of graph-like systems, Bayesian graphs inherently eliminate redundancy and complexity. Bayesian networks possess links to human perception (Jensen, 1996), and when applied to visualising information within GUIs, provide a more intuitive understanding of the data and relationships, even to non-experts (Jensen, 1996).

Both the notion of layers and Bayesian networks have found use within software development, in the form of various layering design patterns (Buschmann *et al.*, 1996) and directed-acyclic graphs or DAGs (Fowler *et al.*, 1999). The present study combines the relaxed layering design pattern of Buschmann *et al.* (1996) with that of Bayesian networks, as depicted in Figure 3.5. In applying layering and Bayesian networks, the data-model used by architects should enforce a graph of directed, acyclic associations amongst concept types, to ensure that instances constructed from these

types in turn hold acyclic and layered references to instances of those concept types. Note also that several related concept types may reside within the one layer, provided they share a common base type of concept—through generalisation. This approach enables the architect to put two different yet related types in the one layer, provided that these types only refer to concept types in lower layers.

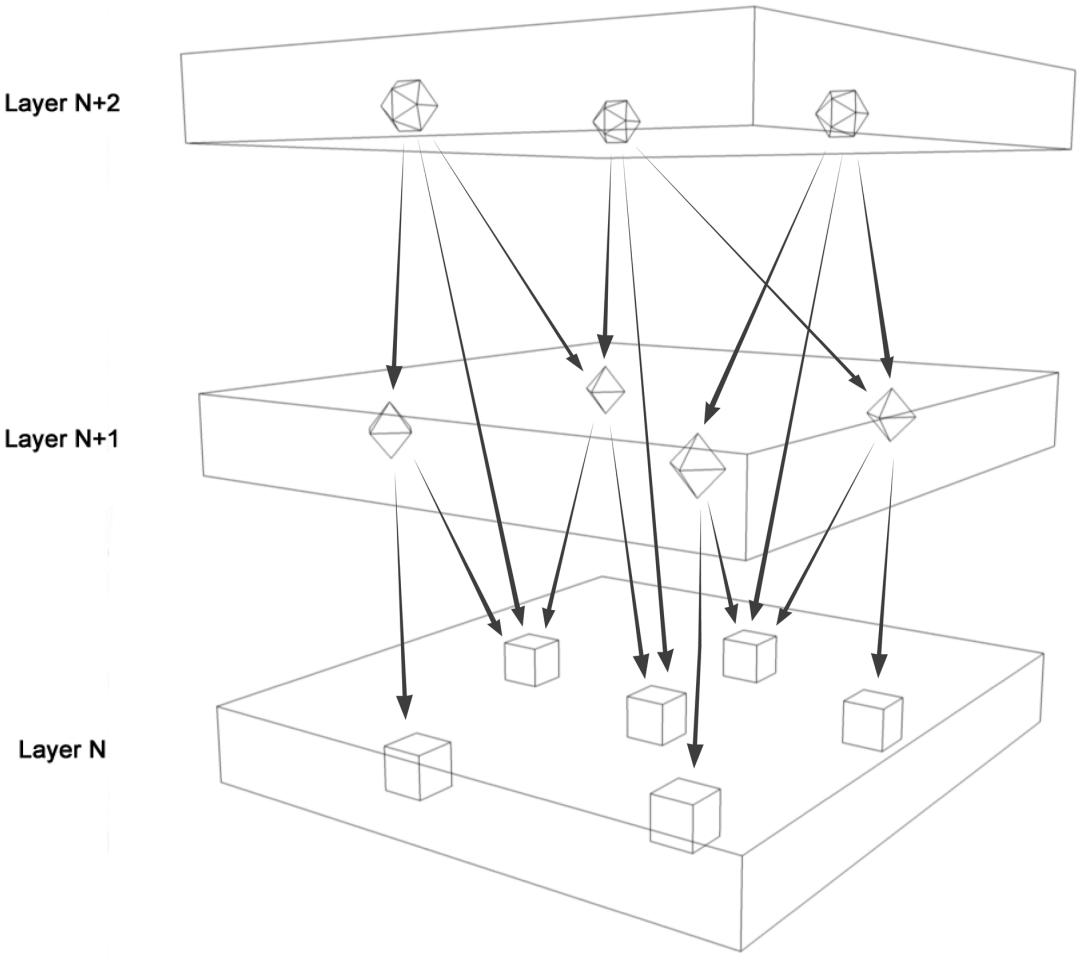


Figure 3.5: A depiction of a relaxed-layered data-model. Instances are depicted as polyhedrons; associations amongst instances are depicted with arrows. Note how instances of a single type reside in the same layer, that associations are directed and acyclic, and that associations can cross layers.

The approach ensures that the data-model, within which practitioners assemble graphs of concept instances, provides an intuitive basis for manipulation and visualisation by the practitioner. A layer consists of instances of the same concept type. Associations kept by the instances within a layer refer to instances of other types within layers below. From here onwards, we shall refer to these lower layers as *sublayers* in relation to the particular layer.

From the practitioner's perspective, each layer is specific to a particular concept type. The practitioner creates instances of a concept type within the related layer, associating it to concept instances upon which it builds. The approach constrains the design space for the practitioner, reinforcing a directed acyclic relationship of associations in the mind of the practitioner—provided that the architect designs their domain ontologies accordingly. The act of creating instances of concept types within specific layers according to Bayesian principles reinforces the practitioner's own mental representations of reality.

To understand the relevance of this approach to simulation assembly, CAPE readers should consider that the classic flowsheet depicts one possible view of an implicitly two-layered data-model. The 2D graph-like presentation of a flowsheet depicts unit operations as nodes, with the material streams as directed edges connecting those nodes. This form of depiction simplifies the underlying data-model, which would implement both the streams and unit operations as nodes. In this case, the stream instances hold runtime references to the related source and sink unit-operation instances. These references produce the directed edges of this alternative graph. If we (a) view this underlying data-model of instances and references as a 2D graph, (b) tilt our view across the plane containing this depiction, and (c) raise all stream nodes out of the plane containing the unit-operation nodes, then we can identify a second layer for stream nodes. Note that these efforts preserve directed, acyclic relationships between streams and unit-operations. Furthermore, consider that we have further concept types that build upon the concept types of the two layers described above. By extending the two-layer case to three or more layers, we have a graphical means of presenting domain ontologies containing three or more concept types. In the case of a finer-grained process-engineering ontology, this approach would enable the visualisation of instances of concept types such as "particulate", "spatial variation", "mass-transfer mechanism", and "heat-transfer mechanism".

These exercises demonstrate Bayesian layering as an effective alternative to the flowsheet. The flowsheet managed to depict unit-operations and streams because it was inherently suited to domain ontologies involving two concept types. Finer-grained concept types were implicitly coded into the mathematical representations for the unit operations and streams. By basing the visual distinction amongst concept types upon layers—rather than according to nodes and edges—we open the possibility to provide the run-time depiction of problems involving three or more concept types, without the need for predefined symbols for those concept types. Thus any concept set may be loaded into the data model for subsequently visualising the assembly of instances of those concept types.

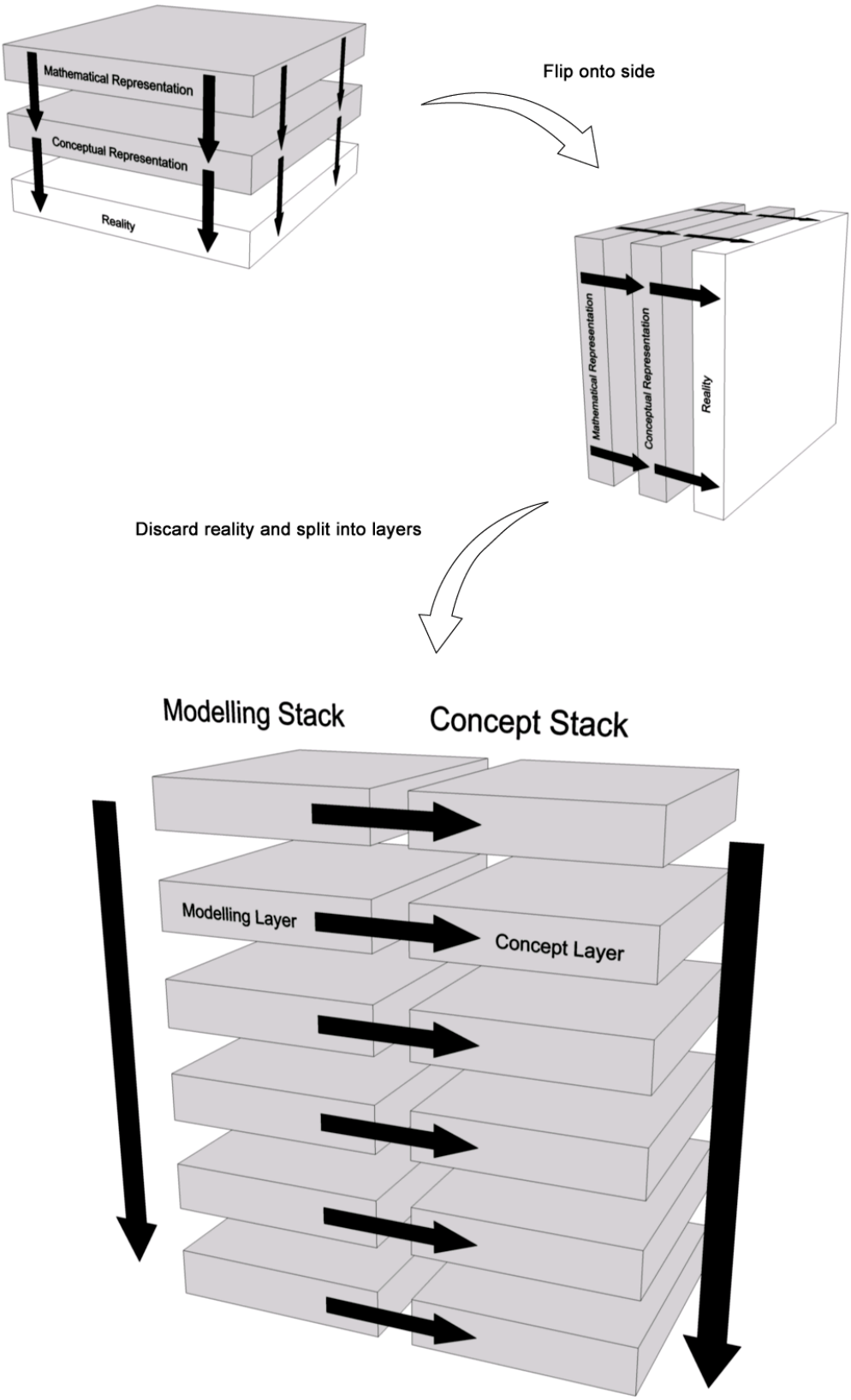


Figure 3.6: Depiction of how the three layer representation of Figure 3.1 can be in turn be layered, allowing run-time assembly of models. Arrows indicate the direction of run-time associations amongst instances within the layers.

Figure 3.6 depicts how Bayesian layering can be applied to decompose the three-layered representation depicted in Figure 3.1. Section 3.1 has already explained that the mathematical technique types must be statically decoupled from the corresponding concept types. Thus, the arrows depicted in Figure 3.1 represent the collective run-time associations from the instances of technique types within the mathematical representation to the instances of concept types within the conceptual representation. By flipping the mathematical and conceptual representations on edge, and ignoring the "reality" layer due to its externality to the computer's representation, we can vertically partition both representations into layers. Because a given technique type is intended for a specific concept type, the layering sequence of the mathematical representation mirrors that of the conceptual representation. Each layer of this counterpart maps directly to a layer of the conceptual representation. The final structure depicted in Figure 3.7 thus maintains a unidirectional partitioning between technique instances of a given layer with the concept instances that they represent.

In summary, this section has rationalised the design of the data-model underpinning the assembly environment for use by practitioners to construct simulations. The data model statically decouples concept types from technique types to allow their independent instantiation and assembly at run-time. The data-model also applies layering and Bayesian network principles to provide intuitive run-time visualisation and assembly of concept instances into in-memory graphs, according to the related domain ontology. These approaches allow the practitioner to load any domain ontology that obeys the Bayesian layering requirements for basing the creation of instances of concept types. This arrangement allows the application of the simulation-assembly environment to multiple disciplines. The next section explains how the combination of layering and Bayesian networks allows the practitioner to express different configurations of the created concept instances that arise within a simulation during solution.

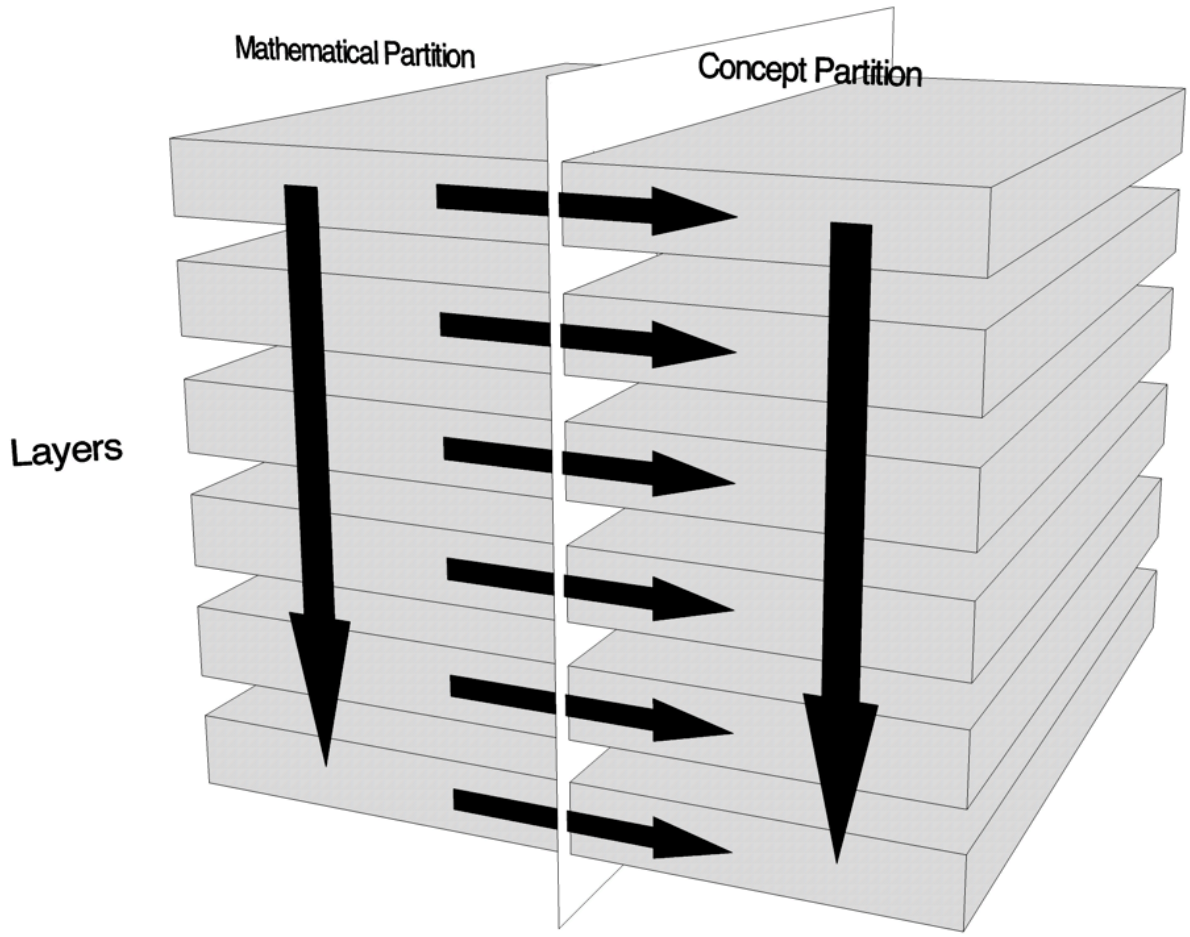
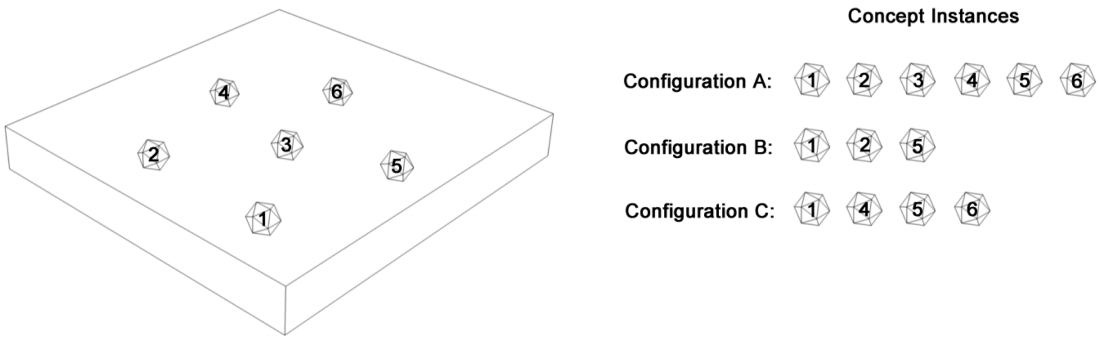


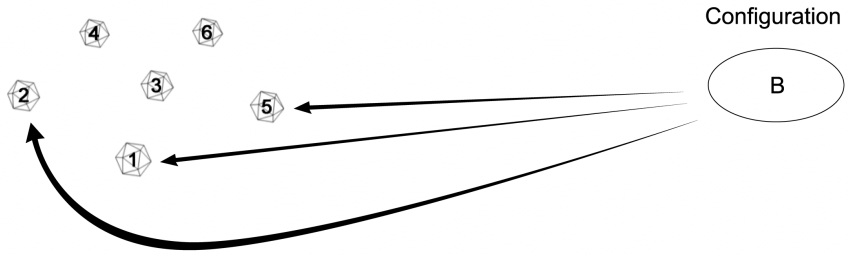
Figure 3.7: Depiction of the final design for assembly-environment's data-model. Arrows represent run-time associations (a) amongst layers and (b) across the partition.

3.6 Configurations

Many simulations produce solutions that mirror changes over some independent variable such as time. A modelling problem's conceptual representation often progresses through different configurations during solution. In other words, from all concept instances recognised to exist in a representation, only a given subset might actually be active at a particular time. A practitioner often wishes to express these configurations when assembling the conceptual representation, to constrain the directions taken by a simulation at *solution time*—the phase of work related to simulation solution.



(a) Concept instances in a layer may belong to multiple layer configurations



(b) Solution: Each layer-configuration instance holds runtime associations to member concept instances

Figure 3.8: Depiction of the grouping of concept instances into configurations.

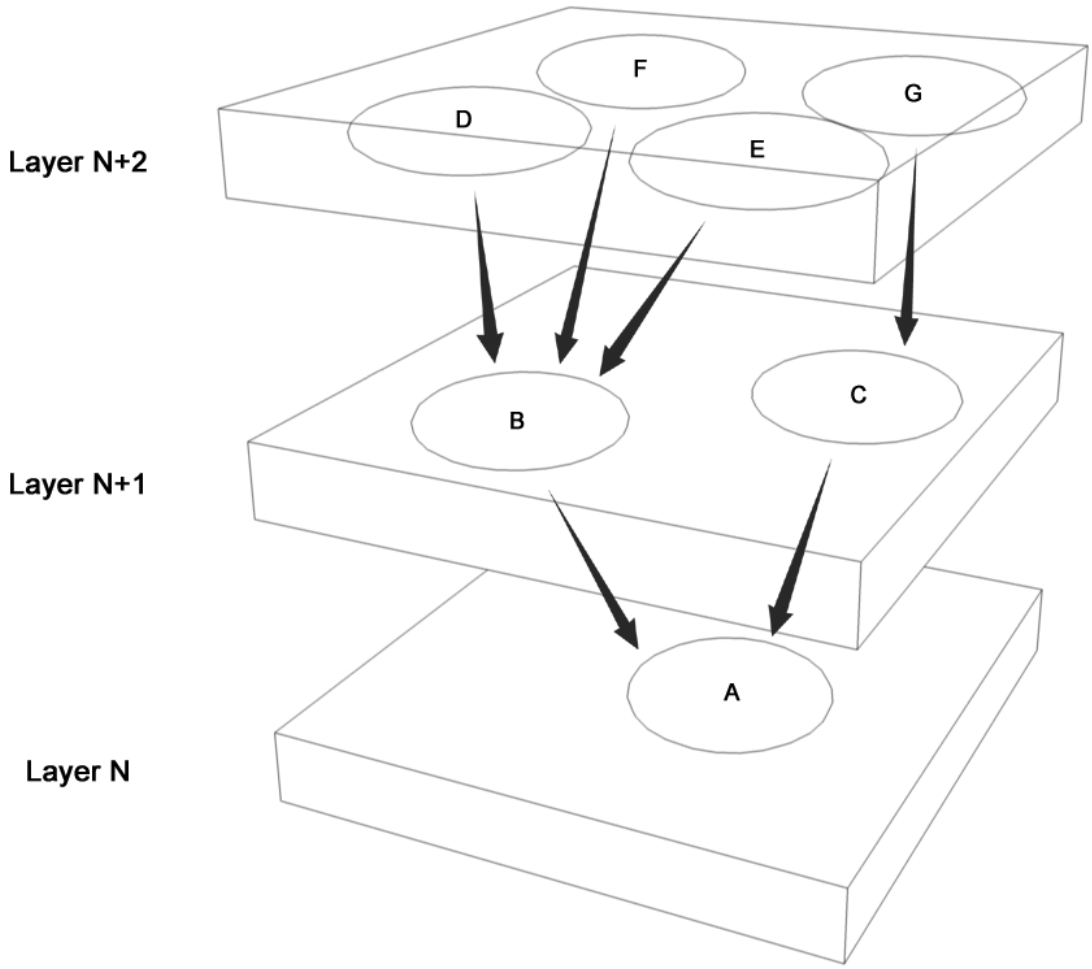
One approach would be to create and destroy concept instances depending on which concept configuration is active at the time. Unfortunately, this approach breaks the constructed graph of run-time associations created during the assembly phase. Additionally, creation and deletion are inefficient actions, particularly as we must re-associate all dependents by rebuilding the association graph of concept instances when changing amongst configurations at solution time.

Alternatively, because each instance of a concept type is unique, the practitioner can specify configurations by associating concept instances with each configuration to which it belongs. This arrangement—depicted in Figure 3.8—would not break the association graph, with each instance of a concept type being able to belong to several configurations. It should be possible to have a concept instance belong to all configurations within the context of a single representation.

This configuration approach and the layered assembly architecture both work very directly on concept instances. In other words, from the practitioner's perspective, both facilities need to be cleanly integrated for ease of use: the goal is to allow valid sets of simulation configurations, while not violating constraints implicit to the provision of a layered architecture.

The solution to this problem involves specifying configurations on a per-layer basis, building these up to represent the overall set of configurations available during the solution phase—as depicted in Figure 3.9. The approach requires the definition of two terms: (a) overall configurations, and (b) layer configurations. An *overall configuration* consists of the subset of all concept instances in a representation that is together active at some time during solution. A *layer configuration* is the set of concept instances within a layer that are together simultaneously active within a specific overall configuration. The design of the last section saw each layer being designated a particular concept type. Hence, layer configurations are meant only for instances of the same concept type.

A concept type depends on those concept types located in sub-layers, due to the ordering of layers. The run-time associations amongst instances of those types are reflected similarly. Sub-layer concept instances may themselves belong to several layer configurations within their own layer. During solution, a subset of all concept instances in a representation might be active simultaneously, according to which overall configuration is active at the time. It follows that when a concept instance within a layer is active, the sub-layer concept instances to which it holds associations must also be active.



Layer Configurations	
Overall Configuration I:	A + B + D
Overall Configuration II:	A + B + E
Overall Configuration III:	A + B + F
Overall Configuration IV:	A + C+ G

Figure 3.9: Depiction of the merging between configuration and layering, to specify overall configurations. Arrows indicate associations amongst concept instances within the layer configurations.

Hence, a constraint exists regarding the construction of the layer configurations: concept instances in higher layers may only target those concept instances in lower layers that are in layer configurations associated to their parent configuration. This situation creates a dependency order amongst layer configurations in successive sub-layers, as depicted Figure 3.9. It follows that configurations in a given layer are themselves dependent on configurations in layers beneath the given layer. Notice that the number of configurations either stays the same or increases with each layer.

This dependency means that each layer configuration is mapped against one layer configuration in the immediate sub-layer—performed at run-time as an association. The dependency implies a restriction on member concept instances of the layer configuration. Remembering that concept instances may belong to several configurations, the targeted sub-layer concept instances must belong to all sub-layer configurations targeted by the layer configurations of the original concept instance. In addition, the targeted sub-layer instances must all belong those configurations: they cannot be split across several layer configurations targeted by the higher layer configuration.

This arrangement creates a constraint: the minimum number of configurations automatically present in a particular layer equals the number of configurations present in the immediate sub-layer. Additionally, the bottom-most layer must have only one available configuration, with all concept instances in that layer belonging to that configuration.

This section has identified the necessary structure for allowing practitioners to specify concept configurations during the assembly of simulations. In the next section we consider how architect-specified domain ontologies can depend on other domain ontologies, due to the dependency of concepts of a particular domain upon concepts of other domains.

3.7 Domain Instances and Domain Ontologies

The work so far has discussed domain ontologies as independent sets of concept types that build upon one another. However, many domains involve concept types constructed upon the concept types of yet other domains. Hence, relationships exist amongst domain ontologies. It follows that, to maintain the Bayesian associations amongst concept types in different domain ontologies, the overall dependencies amongst the involved domain ontologies must themselves be Bayesian—that is, directed and acyclic. Figure 3.10 depicts this situation.

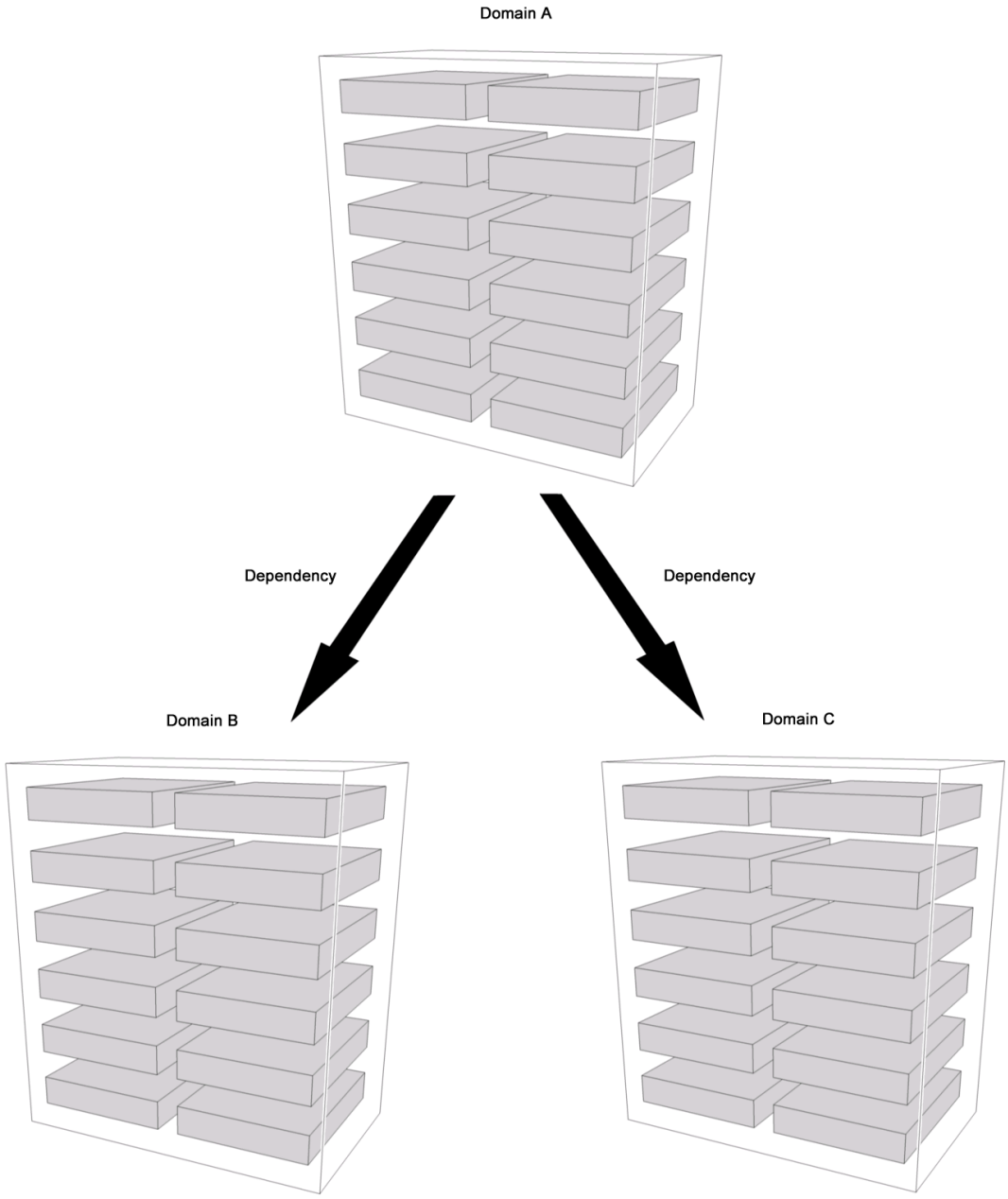


Figure 3.10: Depiction of allowed dependencies amongst domain instances, necessarily directed and acyclic in accordance with the Bayesian relationships amongst the domain ontologies.

Furthermore, just as the practitioner creates concept instances from concept types, a parallel exists for domain ontologies. A domain ontology represents the static definition of concept types, the relationships amongst concept types, and the related layer ordering. The practitioner however uses an instantiation of that domain ontology to construct representations of their simulation problem. This run-time domain instance holds in-memory layers within which practitioners create concept instances to assemble their specific simulation.

To provide context to the practitioner's effort, instances of concept types should be visually depicted within their corresponding domain instances. This visual presentation demarcates the respective domain ontologies.

Just as the Bayesian relationships amongst concept types produces Bayesian associations amongst instances of those concept types, similarly the requirement for Bayesian dependencies amongst domain ontologies produces Bayesian dependencies amongst domain instances. Apart from ensuring continued intuitive visualisation of these domain instances and their related layers to the practitioner, this arrangement allows the continued application of layered configurations—explained in section 3.6—throughout all domain instances.

3.8 Simulation Types and Numerical Solution

As explained in Chapter 1, this study concerns itself primarily with the assembly of simulations, rather than their subsequent solution. However, numerical solution may involve a choice for the practitioner regarding the preferred solution algorithm used to solve assembled simulations. The reader will note that this choice resides within the third tier discussed in section 3.1.

The solution algorithm is often dependent on the type of simulation being performed. For example, simulations of time-varying problems use stepping algorithms to integrate rates over time to produce a time-dependent profile for specific states describing those problems. Hence, prior to selecting the solution algorithm, the practitioner must already have chosen the simulation type.

The choice in simulation type impacts upon the technique types available to the practitioner, because those techniques must be interoperable with the particular numerical algorithm used to solve the assembled techniques. Hence, technique types must be further categorised according to the types of simulation to which they are applicable, for their correct selection by the practitioner.

It follows from these discussions that a run-time means must be available within the assembly environment for the practitioner to choose the simulation type with which they are working. Similarly, a run-time means must be provided to allow the selection of solution algorithms against these simulation types. Because the range of solution algorithms and simulation types is likely to be much smaller than the range of techniques used to represent concept types, the range of choices will most probably be compiled into the tool rather than being accessed from a remote repository.

3.9 Summary

This chapter has designed an infrastructure that addresses the problems discussed in Chapter 2 regarding simulation assembly. The design divides the responsibility of the practitioner into the three roles of architect, developer, and practitioner, to shift software development and knowledge extension practises out of the practitioner's workspace. The design involves distributed repositories that separately store domain ontologies and implemented mathematical techniques. The practitioner assembles run-time representations of their problems as graphs of in-memory instances of concept types. The design of the data-model underpinning the simulation–assembly environment is based around a Bayesian, layered structure to allow intuitive visualisation and manipulation of assembled problems. This approach allows the environment to load any ontology of concept types for instantiation by practitioners. The environment also statically decouples the concept types from their mathematical representations. In this way, practitioners may select, instantiate, and associate technique types to individual instances of concept types at run-time.

Chapter 4: Infrastructure Specification

The present chapter documents a specification for the infrastructure, as rationalised and designed in the previous chapter. The specification presents the infrastructure's types, components, and services in the context to the three roles of architect, developer, and practitioner. As a specification, this chapter neither repeats any design rationale from the previous chapter, nor mandates any particular implementation technologies in the realisation of the design.

4.1 Introduction

The present chapter defines the behaviour and collaborations of the components and types that constitute the infrastructure. The three roles utilise an implementation of this specification in performing their respective work. No inferences are made regarding the choice of implementation technologies: many different implementations could be realised, provided they adhere to the logical/physical and dynamic/static constraints defined in this chapter. Rationale behind the design is not inherent to specification. Such information has been explained in Chapter 3.

The specification applies a discursive style of description combined with UML diagrams. As a specification, the classes presented are types: their presence does not mandate that implementations contain these classes. Implementations need only support the presentation of these abstractions and their related behaviour, so long as they store data to represent state and implement algorithms to support behaviour.

Certain parts of the work are supplemented with Unified Modelling Language diagrams (UML) based on the notation of UML 1.4 (Object Management Group, 2001). Unless indicated, these diagrams are presented from a specification perspective, rather than an implementation perspective (Fowler *et al.* (2000). Hence, unless shown, all depicted classes possess the <<type>> stereotype, with any depicted methods implying behaviour rather than a mandate on implemented method names. Any diagrams stated from an implementation perspective are to provide examples of intent to implementers of the specification. The chapter's text draws attention to entity names with italicised text. For example, *Concept* and *getLayer* represent entities. The reader must differentiate this approach from that used in the UML diagrams, where italics indicate abstract classes and abstract method names.

The chapter consists of three parts, providing (a) a summary of the physical components and their related interactions, (b) a summary of the logical types that recur throughout these components, and (c) an example extension of the infrastructure to discrete-event simulation, to specify the extension mechanisms to be supported. We commence with a discussion of infrastructure components.

4.2 Summary of Components

The infrastructure consists of software components and services that are used by architects, developers, and practitioners to perform their tasks. The following subsections define these components. Each role interacts with the infrastructure at different function points. Additionally, the related tools work with the contents of each repository in different ways. Figure 4.1 depicts the major components and their interactions. Components are presented in terms of their relationship to other components and to types provided by the architecture. The first subsection examines the ontology repository component.

4.2.1 Ontology Repository

An ontology repository provides a database of well-formed ontologies for access by architects, developers, and practitioners. The term "well-formed" implies that the stored ontologies possess a valid structure, allowing their instantiation as domain instances. While the manner of storage implementation is open-ended, the repository must provide the following services:

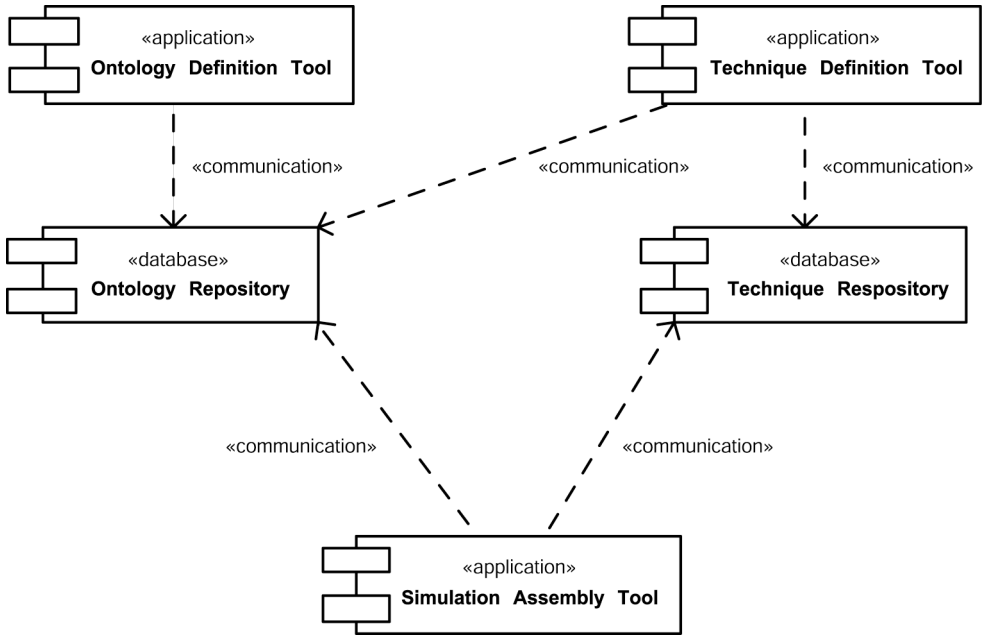


Figure 4.1: Component diagram depicting the major components of the infrastructure.

- Write Service: Architects are the only role allowed to add, modify, and remove ontologies within the repository. Architects perform this task through their ontology definition tool.

- Search Service: All roles are allowed to search ontology repositories based on descriptive keywords. The search returns unique identifiers for ontologies that match the specified criteria. The unique identification scheme is implementation specific.

- Read Service: All roles are allowed to read any ontology from ontology repositories, according to the unique identifier of the ontology.

The repository provides these services for remote use, permitting access by the three roles at different physical locations (nodes)—such as workplaces, companies, and standards bodies. Multiple ontology repositories will exist, each repository located at a different node. The repositories provide a publishing mechanism for remote access by other nodes. This arrangement supports the nature of ontologies: a given ontology might depend on ontologies residing in several repositories.

4.2.2 Ontology Definition Tool

The ontology definition tool provides architects a means to import ontologies into an ontology repository. The tool must support features related (a) the validation of ontologies and (b) the publication of ontologies upon an ontology repository. The tool must also support:

- Specification of the concept type hierarchy and collaborations making up the related ontology.

- Specification of the name, description, version, and unique identifier for the ontology.

- Definition of keywords to assist in searching for the ontology in ontology repositories.

- Validation of the ontology prior to its committal to an ontology repository. A valid ontology possesses concept-type relationships and domain-ontology dependencies that form a Bayesian graph—that is, directed and acyclic in structure. These properties ensure that the ontology may be instantiated within the layering mechanism of the simulation-assembly environment.

The tool must also support the following repository-related features:

- Provide a means for storing a valid ontologies in an ontology repositories.

- Provide a means to search for ontologies in a given remote repository based on descriptive keywords.

- Allow the querying of an ontology repository to list its stored ontologies.

- Allow the reading of an ontology from an ontology repository based on the unique identification system for the ontologies.

The tool needs to be able to perform remote access of ontology repositories. Although not mandated, some sort of version tracking of ontologies would be desirable, as an ontology might well evolve over time as a given standards body refines that ontology.

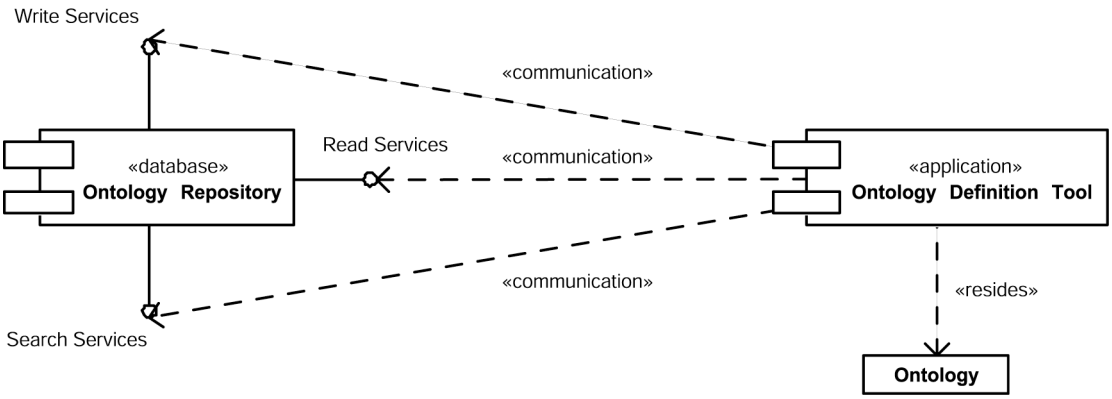


Figure 4.2: Component diagram depicting the interaction of the ontology definition tool with the ontology repository.

4.2.3 Technique Repository

This component behaves similarly to the ontology repository, except that it publishes implemented mathematical-technique types. The retrieval might not necessarily mean the download of the technique as software, possibly allowing remote creation of an instance of a technique type for distributed use. Otherwise, the technique repository provides the same facilities as the ontology repository, stated explicitly below.

The repository must provide the following services:

- Write Service: Developers must be allowed to publish techniques on the technique repository for access by the other roles. Only developers are permitted to perform this task, through their technique definition tool.

- Developers and practitioners must be able to search the repository for available technique types, based on keywords describing the desired technique. The search returns the unique identifiers for techniques that match the specified criteria.

- Read Service: Architects, developers, and practitioners must be able to access, instantiate a remote instance, or download technique types from the technique repository, identified by its unique identifier.

The repository provides these services for remote use, permitting access by developers and practitioners at different nodes. Nodes may include workplaces, companies, the headquarters of

standards bodies, and so forth. Multiple repositories will exist, each repository located potentially at a different node. The repositories provide a publishing mechanism for remote access by other nodes. This arrangement supports the nature of technique types, with their potential dependencies on other technique types: instances of a technique type might build upon instances of technique types residing on several technique repositories.

4.2.4 Technique Definition Tool

This tool provides the point at which developers interact with the overall infrastructure. Developers implement mathematical techniques for concept types of a given ontology. This implementation is written and compiled separately to the technique definition tool: the tool simply provides a means through which developers deploy implemented techniques in the technique repository, thus publishing their presence for access by practitioners. The approach used by the developer to implement the technique is not mandated by this specification.

A mathematical technique could be implemented and compiled by the developer prior to being published in the technique repository for download into the practitioner's environment. However, due the heterogeneous nature of distributed computing—that is, developers and practitioners working on different operating systems and platforms—the compiled technique type might instead be remotely instantiated and executed from the practitioner's environment as a component running on the developer's server. In this case, the technique repository simply publishes the availability of the technique for use.

By being executed remotely, the technique need never be downloaded, so that it may be compiled once in a language chosen by the developer. This option creates greater possibilities for support of distributed, parallel computation, to reduce the time taken when the finally assembled simulation is ultimately solved. The present developments in grid computing would allow such services to provide the implementation behind certain technique types. The technique type would provide a façade to such resources.

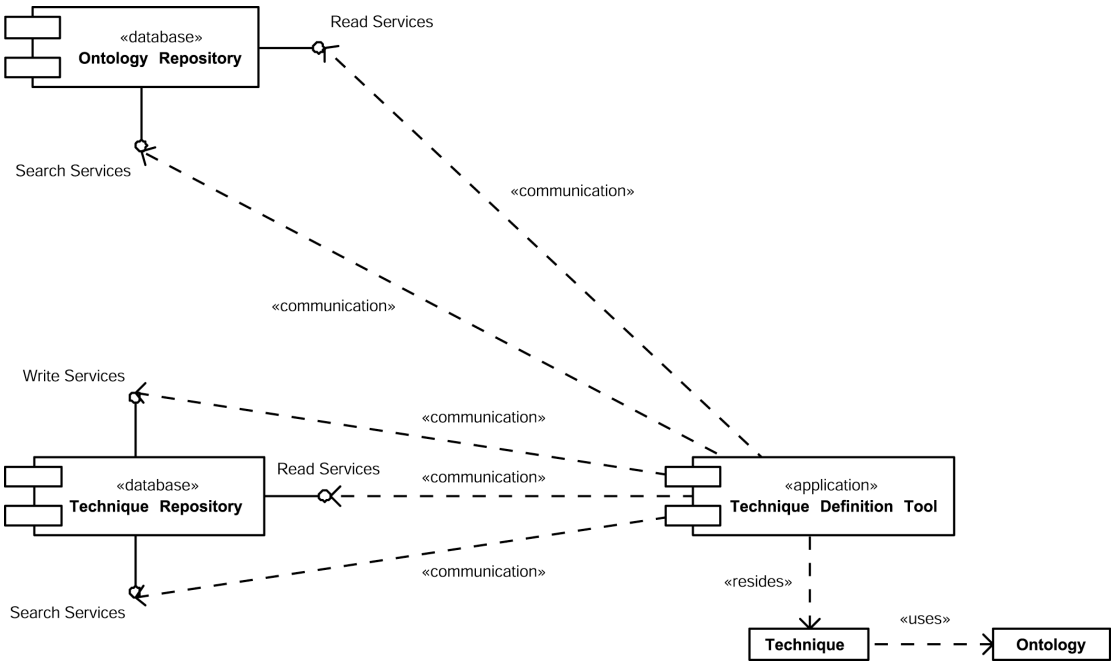


Figure 4.3: Component diagram depicting the interactions of the technique definition tool with the ontology repository and technique repository.

The technique definition tool allows the developer to publish the availability of their written technique through the technique repository. The technique-related features include:

- Allow specification of the name, description, version, and unique identifier for the technique type.

- Allow specification of the concept type and related ontology to which the technique type maps—through the unique identification system used for the ontologies.

- Allow a means for the initialisation of important run-time parameters that shape the behaviour and performance Allow the definition of information describing the ontology to assist searching for ontologies in repositories through keywords.

The tool must also support the following repository related features:

- Perform validation of techniques to ensure that they map correctly to an existing concept type and related ontology, prior to deployment upon the technique repository.

- Provide a means for deploying validated technique implementations upon the technique repository.

- Provide a means of searching for a technique type stored in a repository, based on descriptive information tagged to the technique type.

- Allow querying of a repository to provide a list of the published technique types.

4.2.5 Simulation-Assembly Environment

Practitioners assemble simulations within the simulation-assembly environment. This run-time application allows (a) remote access to ontology repositories and technique repositories, and (b) the run-time assembly of simulations for later solution. The repositories belong to standards bodies, organisations, and software vendors separate to the practitioner. The practitioner uses the assembly environment to search these remote repositories for resources suited to their discipline.

The practitioner first searches ontology repositories to identify ontologies relevant to their field. The practitioner then downloads a suitable ontology. The environment automatically downloads any related ontologies upon which this ontology depends. The assembly environment uses these ontologies to constrain the data model used by practitioners to assemble simulations. The practitioner then searches technique repositories for technique types written for the concept types of that ontology. The tool allows the practitioners to create instances of these technique types for each instance of a concept type within the run-time conceptual representation. Once fully constructed, the practitioner then supplies the assembled simulation to the appropriate solution algorithms to obtain results.

In terms of assembly, the environment presents a Bayesian layered depiction of instances constructed from the concept types. The practitioner assembles graphs of instances of concept types within this layered depiction. The practitioner selects technique types, then instantiates and associates those technique types against individual instances of the related concept types. During assembly, the practitioner graphically selects concept instances to display information related to the associated technique instance. This initialisation involves specifying numeric parameters for the related technique instance. These values and parameters cannot be expressions for parsing, but purely single values. Note that these parameters might be for configuring underlying expressions, previously implemented by the developer.

The environment must support features related to ontology repository access and technique repository access. Ontology-repository features must include:

- Provide a means of searching for ontologies stored in ontology repositories, based on descriptive information tagged to the ontology.

- Allow querying of an ontology repository to list the stored ontologies.

- Allow the download of identified ontologies from ontology repositories.

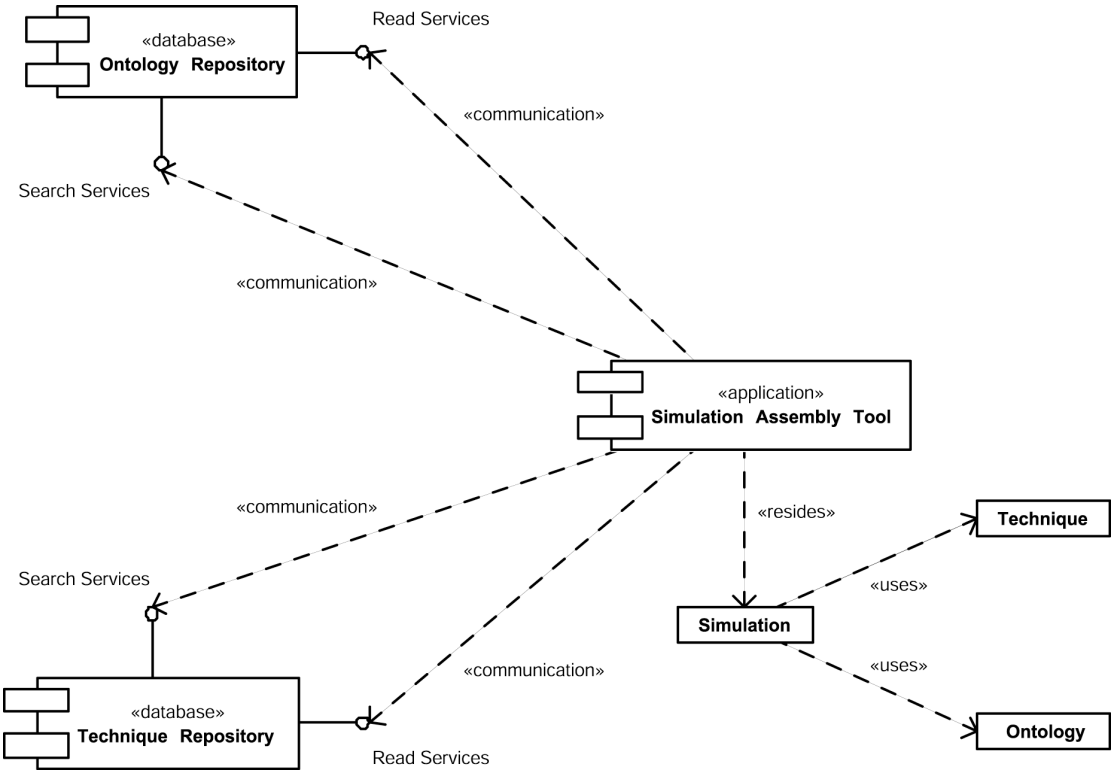


Figure 4.4: Component diagram depicting the interaction of the simulation assembly tool with the ontology repository and technique repository.

Technique-repository features must include:

- Provide a means of searching technique repositories for technique types related to a specified concept type of an ontology.
- Allow the access to identified technique types. Access does not necessarily mean download of the technique type: the technique may be remotely instantiated and executed upon the technique repository.

4.3 Summary of Types

The following subsections each define a single type from the infrastructure. Because the relation amongst these types is not sequential, the subsections are presented in an arbitrary order. The reader must navigate these sections to understand the relationships amongst the hierarchy of types. Each subsection presents more than just a straight-out-of-the-dictionary definition of a term: the title of each subsection is the name of a type to be supported by the infrastructure.

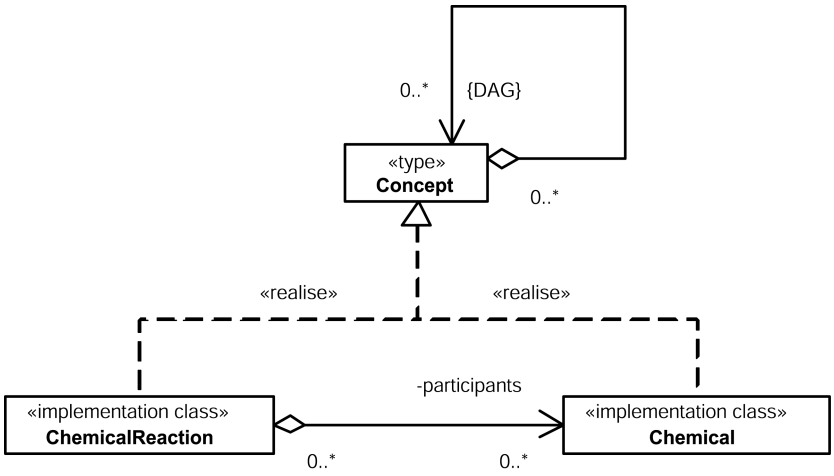


Figure 4.5: Example realisation of *concept* through relation to other concept types, depicted as a static-structure diagram.

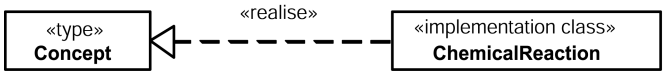
The contents of each subsection explain the expected behaviour of the type, in relation to the overall infrastructure. Each subsection also explains the relation of the type to the roles of architect, developer, and practitioner. This latter discussion describes (a) whether the role uses that type and (b) the manner of this use. We now commence the specification of the various types.

4.3.1 *Concept*

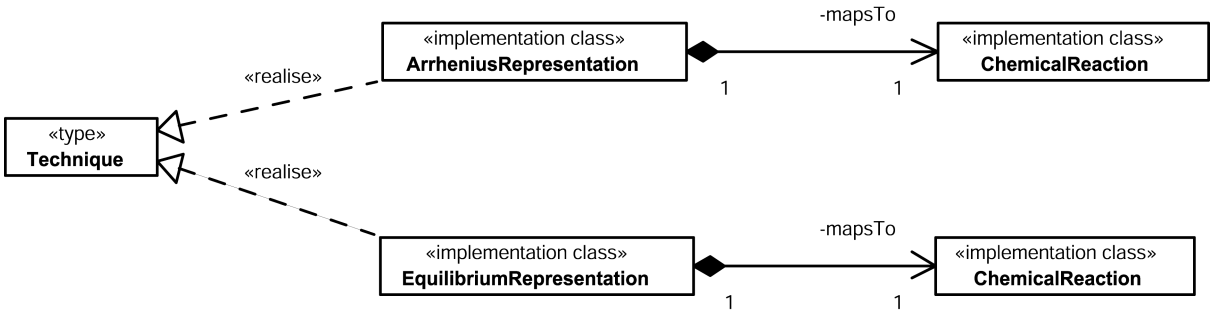
Architects realise *concept* as various concept types. This realisation targets the descriptive portion of these concept types, and contains no mathematical information. Architects realise concept types within the context of a specific ontology. For example, the architect defining a chemistry ontology might decide that a chemical-reaction concept-type resides within that ontology. They then define the behaviour of that concept type in relation to the other concept types defined (a) in that ontology, and (b) in other ontologies. Most concept types are assembled from yet other concept types, themselves realised from *concept* by architects. Figure 4.5 depicts an example of this style of composition for the chemical-reaction concept described above. Figure 4.6 depicts an example of how each role uses *concept*, in this case for a chemical-reaction concept.

Developers do not work with concept types directly. Instead, they realise (implement) *technique* for the mathematical-technique types known to be available for representing a particular concept type. This approach permits mathematical techniques to be written once, then stored in one place for access by many practitioners. The developer does not assemble simulations themselves, as simulation assembly is the role of the practitioner.

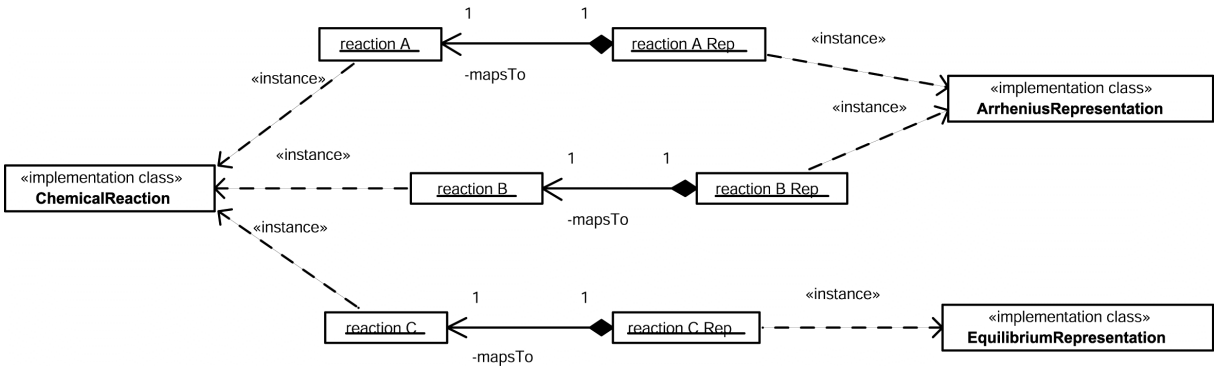
Developers do not work with concept types directly. Instead, they realise (implement) *technique* for the mathematical-technique types known to be available for representing a particular concept type. This approach permits mathematical techniques to be written once, then stored in one place for access by many practitioners. The developer does not assemble simulations themselves, as simulation assembly is the role of the practitioner.Practitioners create instances of realised concept types at run-time to build conceptual descriptions of modelling problems. From the practitioner's perspective, one creates concept instances within the context of the particular layer to which the related concept type is intended. Realised concept instances may be created and destroyed during the assembly and editing of a simulation's conceptual representation: concept instances are never created nor destroyed during solution of a simulation. This approach (a) prevents breakage of the dependency graph used in the initial construction of the simulation, and (b) prevents performance penalties associated to destruction and creation of potentially thousands of concept instances within memory.



(a) Example of the realisation of a *concept* type by an architect.



(b) Example realisations of *technique* by a developer, mapped to the example *concept ChemicalReaction* realisation.



(c) Example of the instantiation of *concept* and *technique* realisations by a practitioner.

Figure 4.6: Analysis perspective static-structure and instance diagrams showing examples of how each role uses *concept* and *technique*.

Concept provides a type for representing any descriptive abstraction whose instances are used at run-time to build conceptual descriptions of problems of interest to the practitioner. Differentiation amongst concept types is implementation specific: the implementation must find its own means of performing concept naming, concept type identification, and the management of the relations amongst concept types. These tasks are not the concern of the practitioner: a practitioner using a simulation assembly tool only understands that the underlying software validates the correct assembly of those concept instances.

4.3.2 *Technique*

Developers realise *technique* as various mathematical technique types to represent instances of a specific *concept* type. An example of this situation is depicted in Figure 4.6(b). Developers realise a technique type by coding the internals of that type, handling all the related implementation issues. This approach permits mathematical techniques to be written once, then stored in one place for access by many practitioners. The developer does not code specific simulations themselves, as simulation assembly is the role of the practitioner. Developers instead code the various mathematical representations for the building-block concept types provided by the related ontology. This approach allows any number of simulations to be assembled by the practitioner at run-time. It should be remembered that a distributed-computing version of a technique is of a different type to the same technique implemented for local computation. A technique instance can access information from technique instances in sub-layers of the related *domain* instance. Likewise, the technique can access concept instances in sub-layers.
The developer must specify:

- The name and description of the technique.
- The intended concept type and related ontology mapped by the technique type, to clearly identify the technique type uniqueness and intended purpose.
- Perform the software-engineering practises to test, maintain, revise, and track versions of the implemented technique.
- Make modifications to the technique, in response to changes to the related ontology by the architects.
- Publish the implemented technique types in a repository for access and use by practitioners.

Practitioners create instances of technique types at run-time for assembly into a mathematical representation that maps the related conceptual description. The practitioner creates an instance of a technique type then associates it to the intended concept instances. A single instance of a realised technique type maps to one or more instances of the related concept type. An example of this situation is depicted in Figures 4.6(c). Practitioners obtain technique types from the technique repository. Within the simulation-assembly tool, the technique instances are stored in the layer of the concept type mapped by the technique. Architects have no involvement with *technique*—neither technique instances nor technique types.

4.3.3 *Layer*

A *layer* groups run-time instances of a specific concept type, providing a container to those instances. Concept types only have meaning in their dependency upon other concept types. Because the infrastructure is based on Bayesian layered structure, concept types of a given layer may only depend on concept types in layers beneath it. Hence, the stacking order of layers indicates to a practitioner the direction of concept type dependencies. Layers thus clarify the practitioner's mind in the assembly of simulations. By thinking in terms of layers, the practitioner is no longer overwhelmed by a confusion of concept dependencies.

The arrangement also assists in the specification of configurations, because resolution into a top-down dependency order simplifies the grouping of concept instances into different layer configurations. Practitioners create instances of a concept type within the layer dictated by the ontology.

Within a given ontology, the architect defines (a) the related number of layers, (b) the particular concept type of each layer, (c) the type of each layer—plain or configurable—and (d) the stacking order of these layers. One concept type exists per layer, with permissible inheritance hierarchies stemming from that concept type. The architect dictates the layer order by specifying the dependency order of the related concept types. Hence only the architect expresses these inheritance trees.

Because developer implementations of technique types map to specific concept types, these technique types are layered in the same order as the related concept types. Developers have no say in the layer order or in the layer types within which concept types reside: they simply map into an existing layer of a specific ontology. Developers need not know if a given layer is plain or configurable.

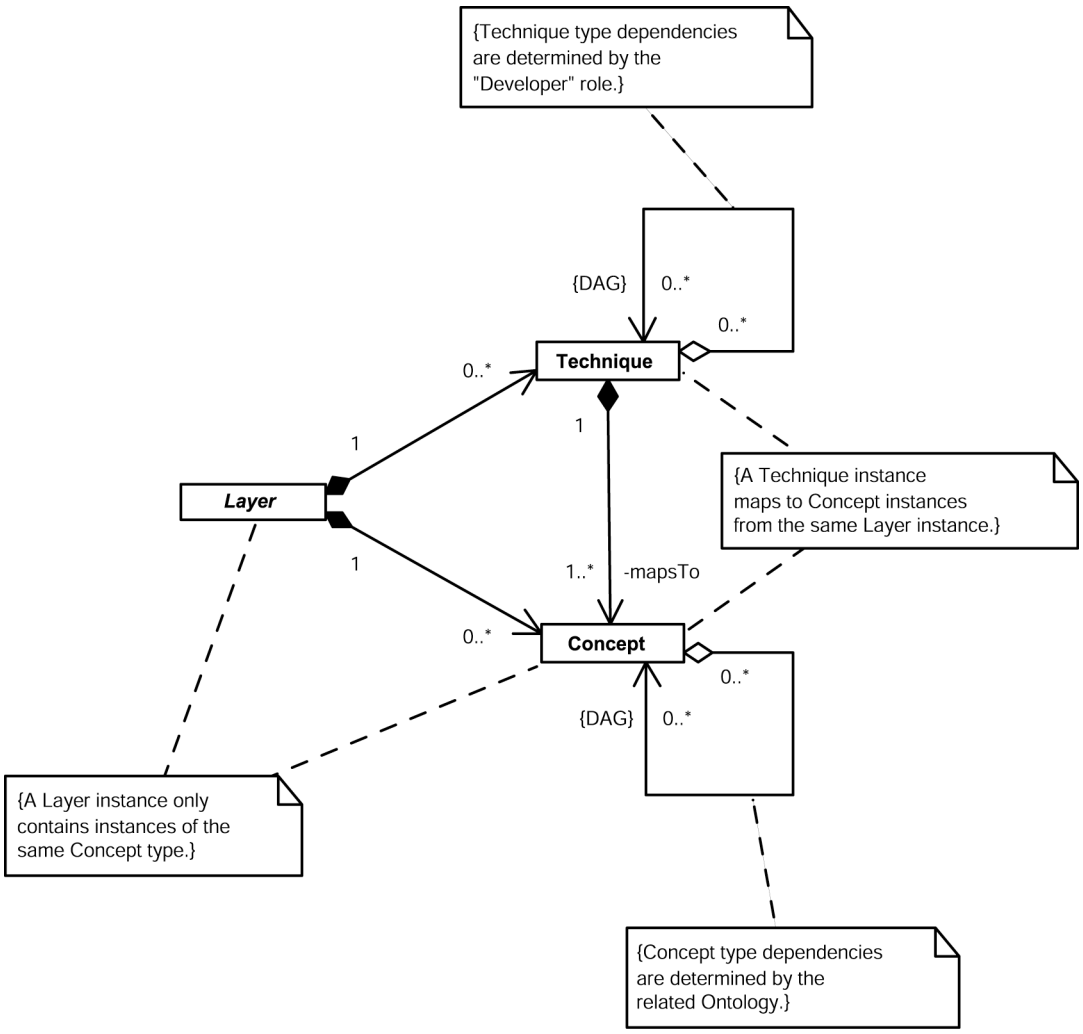


Figure 4.7: Static-structure diagram depicting the relationships amongst *layer*, *technique*, and *concept*.

4.3.4 *Configurable Layer*

A *configurable layer* is a *layer* whose contained concept instances may be assigned to configurations within the layer. The architect is involved with the *configurable layer* at the ontology level: the architect defines in the ontology if a particular layer is of a *configurable layer* type. Developers do not work with *configurable layer*. Practitioners simply use a given *configurable layer* instance to create layer configurations into which concept instances from the related layer are added.

4.3.5 *Plain Layer*

A *plain layer* is a *layer* within which practitioners cannot create *configuration*s. In other words, all *concept* instances within the layer are present within all *overall configuration*s of the assembled conceptual description—rather than being partitioned like those belonging to a *configurable layer*. An implicit rule arising from this arrangement is that no *plain layer* may exist above a *configurable layer* within a given *domain*. This law applies specifically to the architect developer, who defines the ontology for the particular domain. Client developers and practitioners automatically obey this rule as they are restricted by the ontology predefined by the architect.

4.3.6 *Configuration*

Configuration is a group of concept instances collectively active at solution time when the related *configuration* is active. Technique instances have no place in configurations, because a technique instance is only active when the related concept instances are active. Architects have an indirect involvement with *configuration*: they define if a particular *layer* is configurable or plain. Developers have no involvement with *configuration*. Practitioners may create configuration instances within configurable layers. These layer configurations assemble into overall configurations.

4.3.7 *Layer Configuration*

Layer configuration is a configuration of concepts within a layer. Hence, all concept instances within a layer configuration are of the one concept type, because a layer can only contain instances of the one concept type. A given instance of a concept may be added to one or more layer configurations. These layer-specific configurations assemble into large-scale configurations. By taking this approach, a concept instance may be shared amongst several large-scale configurations, indicating those concept instances that are active when a particular layer configuration is active. In this manner, concept instances are neither created nor destroyed at simulation-solution time; instead, different concepts become active or inactive when a simulation switches between different configurations. This approach prevents breakage of inter-layer concept-instance references, because concept instances need not be destroyed nor created to swap between large-scale configurations.

Neither architects nor developers work with *layer configuration*. Practitioners create *layer configuration* instances within configurable layers. The practitioner then adds concept instances from that layer to the desired *layer configuration*s, according to the simulation problem being expressed. A concept instance may belong to multiple *layer configuration*s. The practitioner assembles these configurations at *assembly time*—the phase of work related to simulation assembly. The configurations become active at various periods at solution time.

4.3.8 *Overall Configuration*

Overall configuration classifies the cumulative assembly of all concept instances in the conceptual representation of a simulation. Neither architects nor developers work with *overall configuration*. Practitioners only indirectly create overall configuration instances because they create layer configuration instances that are automatically assembled by the simulation-assembly tool into overall-configuration instances. The simulation-assembly tool determines the available overall configurations by analysing the permutations of the created layer configurations.

4.3.9 *Domain*

A *domain* is a collection of *layer* instances, stacked in an order dictated by the related *ontology*. Architects define the layer order of the domain in the related ontology, because of (a) the concept type they assign to each layer, and (b) the interdependencies amongst these concept types is directed and acyclic. Developers have an indirect involvement with *domain*: mathematical techniques realised by developers slot into specific layers of the domain, a consequence of each technique type targeting a specific concept type.

Practitioners work within *domain* instances at run-time: they use a domain instance to access the layer of interest to create instances of the desired concept type. Domains are never directly created nor destroyed by the practitioner: instances of a *domain* are created when a practitioner specifies that a *simulation* instance be created from a particular *ontology*. Multiple *domain* instances may exist within the one *simulation*, with one *domain* instance per *ontology* instance within that *simulation*. Multiple domains within the one simulation arise from dependencies of a particular ontology upon other ontologies. Figures 4.8 and 4.9 depict instance diagrams of such example situations.

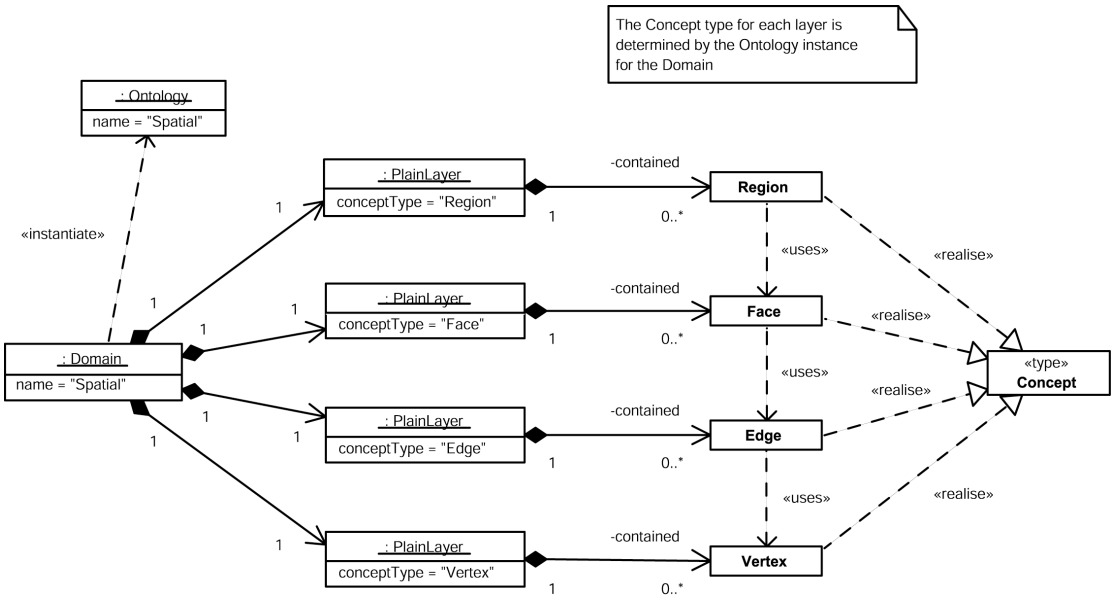


Figure 4.8: Instance diagram depicting an example of the run-time relationships amongst *ontology*, *domain*, *layer* (*plain layer*), and *concept*.

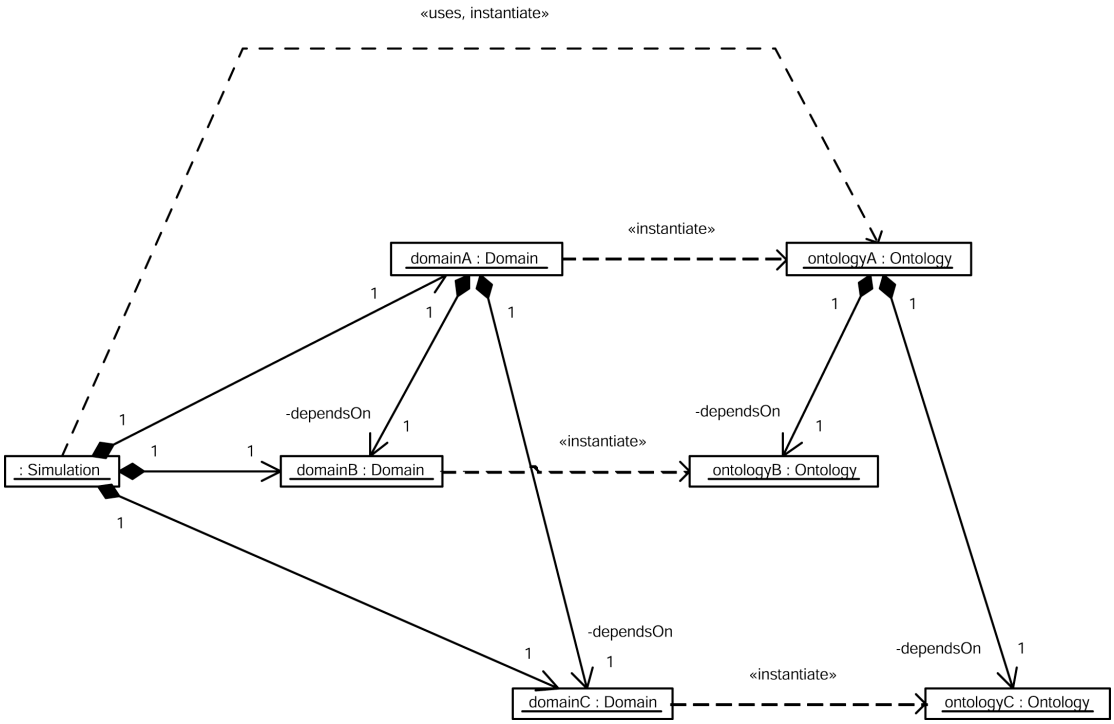


Figure 4.9: Instance diagram depicting a run-time example of how *simulation*, *domain*, and *ontology* relate.

4.3.10 *Ontology*

An *ontology* provides a metaclass to the conceptual description provided by *domain*. Architects define ontologies by specifying the following information:

- The gamut of concept types specific to a particular discipline.
- The rules governing the assembly of those concept types.
- The relationships amongst concept types.
- The layer ordering for those concepts.
- The type of each concept layer—plain or configurable.
- The other ontologies the particular ontology depends upon.

An ontology also indicates any dependencies on concept types defined in other ontologies, as dictated by the dependencies of contained concept types. Note that an *ontology* contains no details of mathematical models.

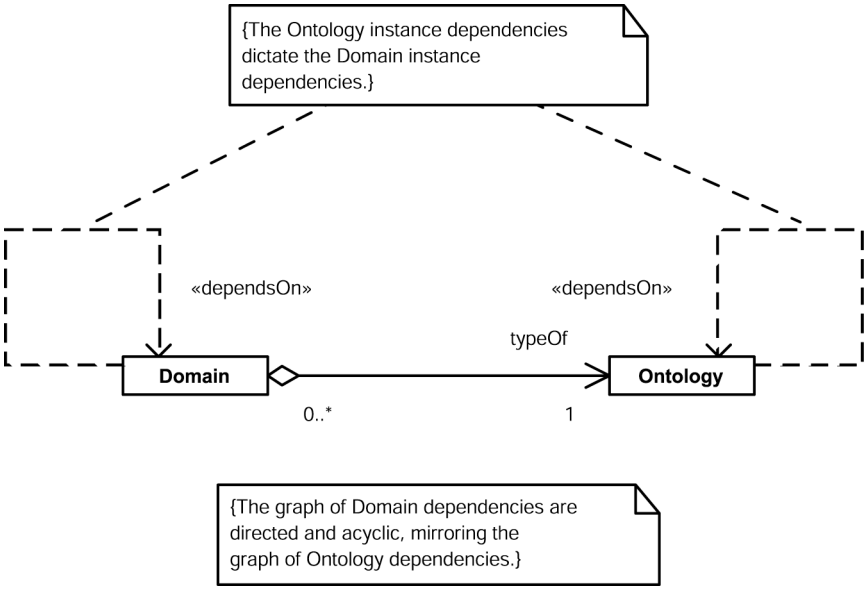


Figure 4.10: Static-structure diagram depicting the relationship between *domain* and *ontology*.

A practitioner, using their simulation-assembly tool, only understands that they select the particular ontology with which to work. The tool creates a *simulation* instance from the selected ontology. The practitioner then assembles a description of their problem within that *simulation* instance using instances of the concept types provided by that ontology. A practitioner understands that the underlying software validates the correct assembly of those concepts. Enforcement of type checking and type relationships is specific to the implementation, and irrelevant to the concerns of the practitioner.

This arrangement has consequences for developers and practitioners, because their techniques and simulations depend on written mathematical techniques and assembled simulations based on that ontology. Hence, version control is important: architects should choose wisely the frequency and magnitude of change upon ontologies. With the concept types defined within an ontology, neither developers nor practitioners alter the relationship of the concept types. Hence, the leaves of the inheritance tree of an ontology are concrete and not intended for further inheritance by the practitioner or developer, because these other roles do not create concept types. Once the concepts are defined they are intended for use. Neither developer nor practitioner touches the inheritance hierarchy. This arrangement produces the leaf requirement expressed earlier for *concept*.

Developers use the ontology to identify the concept types and the related ontology for which they are coding mathematical techniques. Developers have no involvement in modifying the ontology: they only write mathematical techniques for the concept types within the ontology.

Practitioners do not modify ontologies. Practitioners select the ontology they desire to work with within a running instance of the simulation-assembly tool. This selection configures the tool to provide familiar concept types for use by the practitioner in assembling the conceptual representation of their problem. For example, a process engineer would select the process-engineering ontology. The ontology specifies how the tool should enforce rules controlling the assembly of instances of those concept types. This approach prevents degenerate assemblies of the concept instances, which would invalidate the representation, breaking the rules of assembly for the concept types of the practitioner's domain.

4.3.11 *Simulation*

Realisations of *Simulation* provide the various simulation types known to exist. This realisation is currently outside the responsibility of the architect, developer, and practitioner: only infrastructure maintainers realise these simulation types. A valid argument could be posed to make *simulation* a form of *concept* realisable by architects. The present specification does not as yet consider this option. Such a facility would require entities from the present specification to be referable from ontologies loaded into the assembly environment—the classes defined in this specification would need to be concepts defined in their own ontology. Support for such a complex recursive facility has for the moment not been considered, to provide a specification that can be implemented within a single study.

Simulation consists of one or more *domain* instances collectively dependent in a directed, acyclic graph of associations—an example has been depicted in Figure 4.9. *Simulation*, along with management entities for associating stages, configurations, directives, conditions, and timelines. A Simulation is allowed to map against only one ontology. However, different Simulations can be connected providing a bridging amongst different concept sets of different ontologies, in a way providing a sense of interconnecting low-level miniscule concept ontologies with larger general concept bridges.

Practitioners select what an ontology given the spectrum of concepts being used. They use the approach to navigate/select the layer they intend to work with. Select the stack of layers in their simulation corresponding to the domain of concept they wish to work with from moment to moment. For example, a practitioner might wish to work within the chemistry domain-related portion of a simulation. They descend into the representation and select the particular stack corresponding to that domain—for a given representation instance there is one stack per domain within that representation, providing uniqueness. They would then focus on the layer related to chemical reactions, and proceed to create, remove, or query concept instances (i.e. chemical reactions) in that layer.

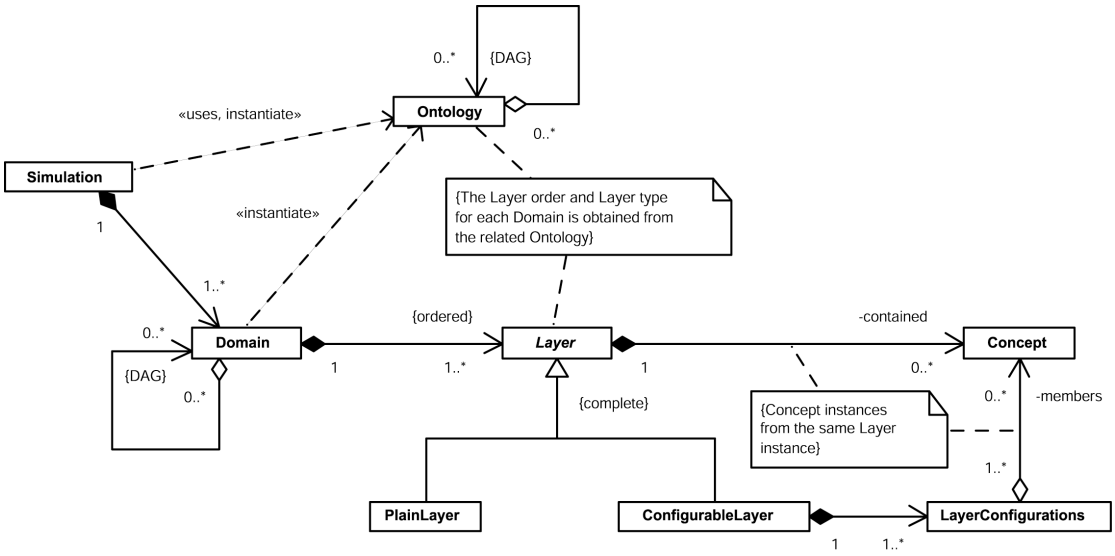


Figure 4.11: Static-structure diagram depicting the relationships amongst *simulation*, *ontology*, *domain*, *layer*, and *concept*.

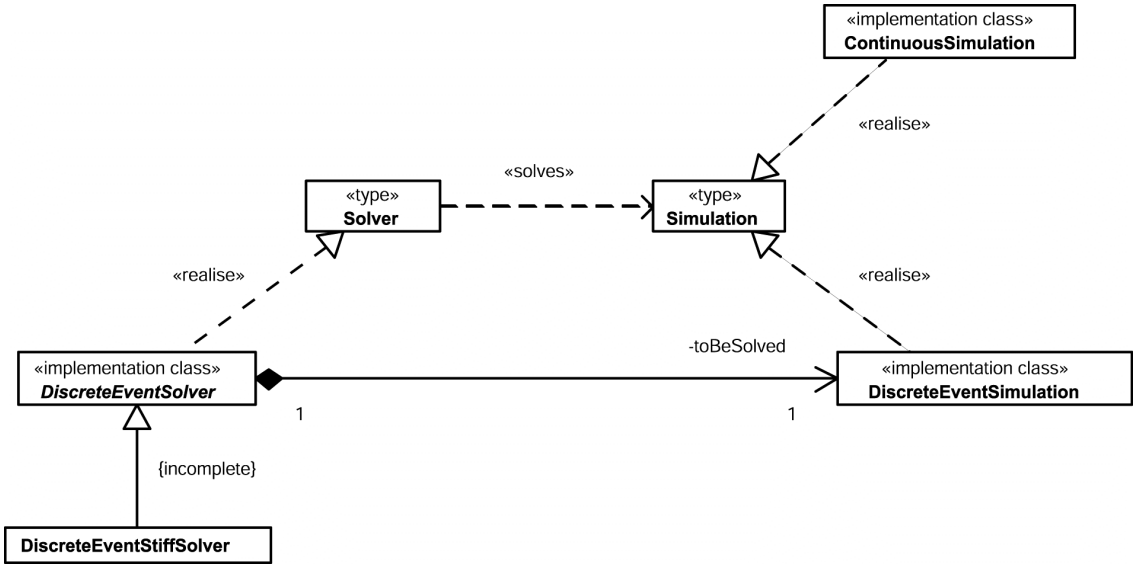


Figure 4.12: Static-structure diagram depicting (a) the relationship between *solver* and *simulation*, and (b) example realisations of these types to show their usage by the infrastructure implementers.

4.3.12 *Solver*

Realisations of *solver* provide the solver types that exist to drive the solution of assembled simulations. The realisation of *solver* is currently outside the responsibility of the architect, developer, and practitioner: infrastructure maintainers only realise these *solver* types. A valid argument could be posed to make *solver* a form of *technique* realisable by developers. However, this would need a similar recursive facility as described by Section 4.3.11 for making entities within the present specification referable from techniques used by the assembly environment. As discussed in Section 4.3.11, support for such a complex recursive facility has for the moment not been considered, to provide a specification that can be implemented within a single study. Figure 4.12 depicts the relationship amongst *solver* and *simulation*, as well as the relationship between *solver* and implementations of *solver*.

4.4 Summary

This chapter has specified the components and types making up the proposed infrastructure. The infrastructure consists of five components: (a) the ontology repository, (b) the technique repository, (c) the ontology-definition tool, (d) the technique-definition tool, and (e) the simulation-assembly environment. These components may reside at different physical locations or at the same physical location. Various types within the infrastructure reside at different components. This choice of components and types divides responsibilities amongst the roles such that practitioners can perform code-free expression of their simulation problems. In the next chapter, we explain how this software specification was implemented.

Chapter 5: Software Implementation

Previous chapters have identified the sofware components necessary for the desired system. The present chapter describes the choices used in this study to implement the software based on that system. This implementation is intended for demonstrating validity of the proposed design, rather than providing a high-performance production system. Source code for the implementation is available at https://github.com/williamwaterson/protolayer.

5.1 Essential Components

As identified in the previous chapter, an implementation of the proposed system consists of the following software components:

Technique Repository: a server that publishes computational techniques available for solving subportions of assembled simulations submitted to the overall computational network.

Ontology Repository: a server that (a) stores and catalogues known ontologies, (b) catalogues available techniques which implement concepts within those ontologies, (c) stores the IP address of technique repositories and related techniques distributed across the network, and (d) allows searching of all information stored within the repository.

Simulation-Assembly Tool: an application that allows users to (a) graphically assemble conceptual representations of their problems, (b) search the remote central repository for ontologies and techniques relevant to their discipline, (c) configure the client with downloaded ontologies, and (d) execute assembled simulations on remote computational resources.

Ontology Definition Tool: An administrative tool for starting and stopping the ontology repository. The tool is also used to add and remove ontologies within this repository.

Technique Definition Tool: An administrative tool for starting and stopping the technique repository. The tool is also used to deploy and undeploy already compiled techniques within the technique repository.

5.2 High-Level Choices

The software was implemented in C++ and the source code was compiled with the GCC compiler to produce executable binaries. Development proceeded on Mac OS X. The UNIX-like aspects of Mac OS X enabled implementation of server components in the software; while the user-interface libraries of Mac OS X enabled implementation of the graphical assembly environment.

5.3 User Interface Implementation

The user interface for the graphical assembly environment was developed using the Apple Carbon libraries available on Mac OS X. These libraries provide graphical controls—buttons, text fields, radiobuttons, and other such controls—displayed by the application.

The OpenGL libraries were used for rendering three-dimensional scenes, with the management of scene information performed with a scene graph implementation. The scene graph hierarchically groups vertices, edges, and faces according to the onscreen objects they represent. This approach allows manageable propagation of orientation information—position, rotation, scale—for grouped spatial elements within scenes.

Spatial-orientation information was propagated throughout the scene graph using quaternion transformation, an approach enabling matrix-vector operations to recalculate vertex positions. The necessary management of topological relationships amongst regions, surfaces, edges, and vertices was implemented as a boundary-representation (brep) spatial model. The definition of curved surfaces and curved edges in these models was implemented with recursive subdivision techniques.

All text rendered in 3D scenes was displayed using the bitmap font support provided by OpenGL. In this way, regular 2D fonts from the underlying operating system are used to display onscreen text in 3D scenery. Also, a cel-rendering algorithm was used in the 3D-scene rendering process to provide an outlined visual appearance for the depicted concept-instance nodes. The visual appearance of the resulting 3D-layered rendering can be seen in the screenshots of Chapter 8.

5.4 Server Implementation

The networking aspects of the system were implemented using classic UNIX-oriented techniques: TCP/IP socket libraries were use to provide reliable network connection management; XDR encoding libraries were used to provide bit-order neutrality for data transfer amongst different computing hardware architectures. Both TCP/IP and XDR related libraries are available on many UNIX-like implementations, making the code portable to other UNIX-like operating systems. An internal protocol was implemented to encode and decode requests in the communications amongst the client and server components. The implementation of this protocol was incorporated into the graphical assembly client to enable requests to the remote central repository and remote technique repositories. Implementations of these protocols were similarly incorporated into the two server components to process incoming remote requests from the client.

Being servers, the technique repository and ontology repository potentially handle multiple parallel requests from different running instances of the client. The server implementation used the POSIX Thread libraries—available on many UNIX-like operating systems—to manage the resulting multiple threads of execution, necessary for processing parallel requests from clients. Critical sections of code that necessarily require a single thread of execution to prevent race conditions amongst threads, were protected with the mutex-locking facilities of the POSIX Thread libraries.

Multithreaded execution also requires management of time-varying memory demands placed on the servers, as the demand of parallel requests from many clients may also be time-varying. An efficient solution is to reuse existing in-memory objects to avoid the performance overhead of simply creating and deleting in-memory objects at the beginning and end of each client request—a time consuming process. A basic pooling algorithm was implemented within the servers to handle this requirement.

The ontology definition tool and technique definition tool were implemented as commandline administrative tools for UNIX. Both tools used the UNIX process management facilities to start and stop server instances. The ontology definition tool manipulates SQL tables within the database for storage and searching. The technique definition tool was implemented to accept sharable libraries produced by the GCC compiler. The necessary dynamic loading of precompiled techniques by the technique repository was implemented using the dynamic loading facilities of Libtool.

5.5 Data Storage

Data storage for the ontology repository was provided by a third-pary relational database, in this case, mySQL. The repository searches the database through submitted SQL queries. The C++ implementation uses the mySQL C libraries to connect to the database and to submit requests.

5.6 Further Findings Discovered During Implementation

During the implementation of the software, two problems became apparent to which the author proposes solutions here. The issues are:

1. The practitioner role needs to be able to configure the numeric parameters of techniques downloaded into the simulation-assembly environment and mapped to concept instances within the visualisation system, and all performed at runtime within the simulation assembly environment.

2. The selection of the canonical programming language to be mandated by the implementation and enforced upon the developer role needs to be more open ended than simply requiring the use of C++—as used in the implementation within this thesis. Additionally, the compiled binary libraries loaded into the technique repository need to be executable across multiple architectures and operating systems, for interoperability.

These problems were realised late in the study, and the solutions described here are not presently incorporated into the implementation—essentially, future work. The solutions to these problems are as follows.

With respect to the first problem, the author proposes that an additional requirement be introduced. For the practitioner to configure the numeric parameters of techniques mapped to concept instances within the 3D-layered visualisation, the practitioner must be able to click upon the nodes within that 3D-layerd visualisation to display an associated dialogbox for the fields whose numeric values can be set for each technique instance. Because the developer role is responsible for implementing mathematical techiques, the developer role must also define the visual layout of this dialogbox for each mathematical technique. This visual layout information must be coupled to the mathematical technique itself, for collective upload into the technique repository.

It follows that the technique repository itself must store this visual layout information with the technique implementation, both for download by the practitioner's simulation-assembly environment. The selection of technology to achieve this functionality has not been chosen at the time of writing of this thesis. However, at this time of writing, visual definition technologies such as XUL and HTML5 are known to be becoming popular for distributed-system GUI layout definitions.

With respect to the second problem, the author notes that recent technologies now enable greater interoperability of binaries and libraries written in disparate programming languages. In particular, the intermediate language representation (IR) facilitated by the LLVM Compiler Infrastructure—see http://llvm.org—alleviates this problem. This technology permits developers to compile code from a multitude of commodity programming languages into the neutral intermediate representation (IR) of LLVM, for either subsequent execution within a universal virtual machine, or for reverse export back into other commodity programming languages. The author proposes that a subsequent implementation of the system that stores the mathematical techniques as LLVM IR in the technique repository—rather than the GCC compiled C++ shared libraries described earlier in this chapter—would provide the much needed interoperability. It would also allow multiple commodity programming languages to be used by the developer role, providing greater flexibility. This approach would necessarily require the LLVM virtual machine to be embedded into the practitioner's simulation assembly environment, to execute downloaded LLVM IR implemented techniques. Fortunately, the LLVM project is provided under very friendly open source licences, making incorporation of the LLVM virtual machine into the simulation assembly environment even more agreeable.

The author again highlights that, due to the constraints of a single PhD research thesis, the findings discovered in this section are not yet included in the implementation presented by this thesis. These are findings discovered by the thesis in going through the experiment of implementing the proposed simulation assembly infrastructure for the first time. These are recommended areas for future work.

5.7 Summary

This chapter has outlined the choices made to implement the proposed system. The chapter specified the third-party technologies used to implement the software. The chapter also stated which portions of the system were coded by the author. The next chapter develops an example ontology providing detail of how an architect would identify concept types from a given problem domain.

Chapter 6: Example Ontology Analysis (Process-Engineering)

To clarify the intent of the specified infrastructure, the present chapter designs an example process-engineering ontology, constrained by the directed, acyclic, relaxed layering requirements inherent to the infrastructure. This effort demonstrates how an architect decomposes the mental constructs for a particular discipline. The work culminates in the specification of this example process-engineering ontology in the next chapter.

6.1 Introduction

As explained in Chapter 3, architects design domain ontologies to satisfy developer- and practitioner-related needs. The present chapter demonstrates how the architect performs this task, by designing an example ontology for the discipline of process engineering. To perform this task, the work must identify the following features:

- the dependencies of the specific domain ontology upon existing domain ontologies,

- the concept types within the domain ontology that will be instantiated by the practitioner at run-time and statically bound against by developers to implement technique types.

- the static and dynamic relationships amongst these concept types, while satisfying the Bayesian layering requirements of the infrastructure—both amongst concept types and domain ontologies.

To satisfy the reader's curiosity, the application of these steps by the present chapter produces the concept-type layer order depicted in Figure 6.1 and the ontology-dependency order depicted in Figure 6.2. A reader seeking a more formalised, detailed specification of relationships amongst the concepts of the resulting process-engineering ontology, is directed to Chapter 7—the result of the analysis performed in this present chapter (Chapter 6).

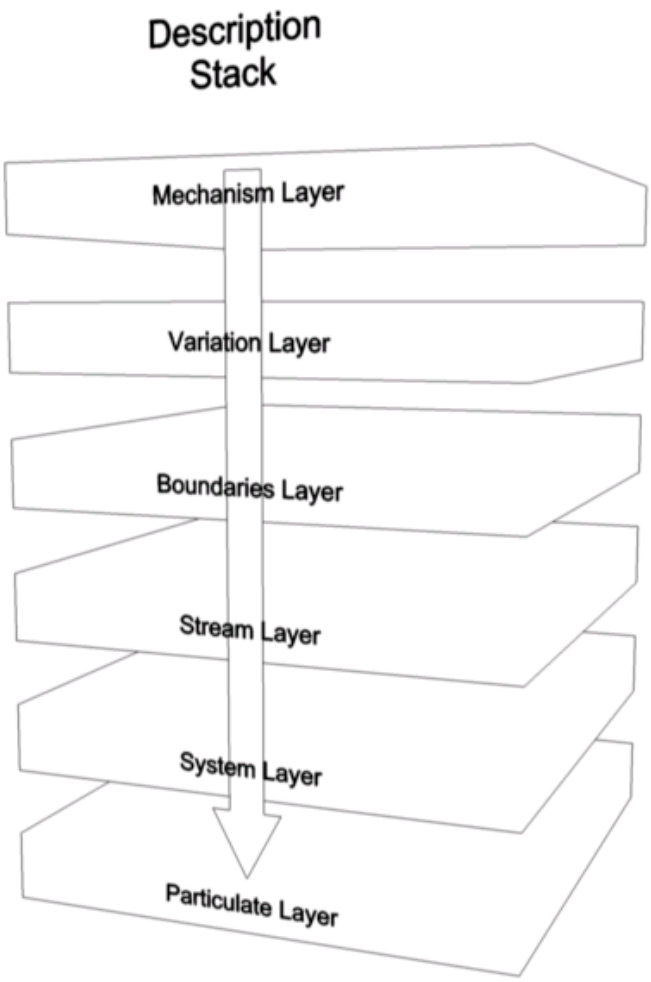


Figure 6.1: Layer order for the process-engineering domain ontology.

The reader will note the dependency of the process-engineering ontology upon the chemistry and spatial ontologies. In this chapter, development of the spatial ontology shall be left to a future work: the present chapter is interested in the process-engineering ontology specifically, and a complete analysis and development of such a spatial ontology is potentially an entire study unto itself, due to the complexities involved. To this end, the analysis shall be focused upon the process-engineering ontology, with a brief summary of the chemistry ontology in Section 6.2—due to its appearance in the demonstration walkthrough presented in Chapter 8. The reader might wish to skip to section 6.3 if they wish to focus upon analysis of the process-engineering ontology specifically. The chapter commences with a summary of the chemistry ontology.

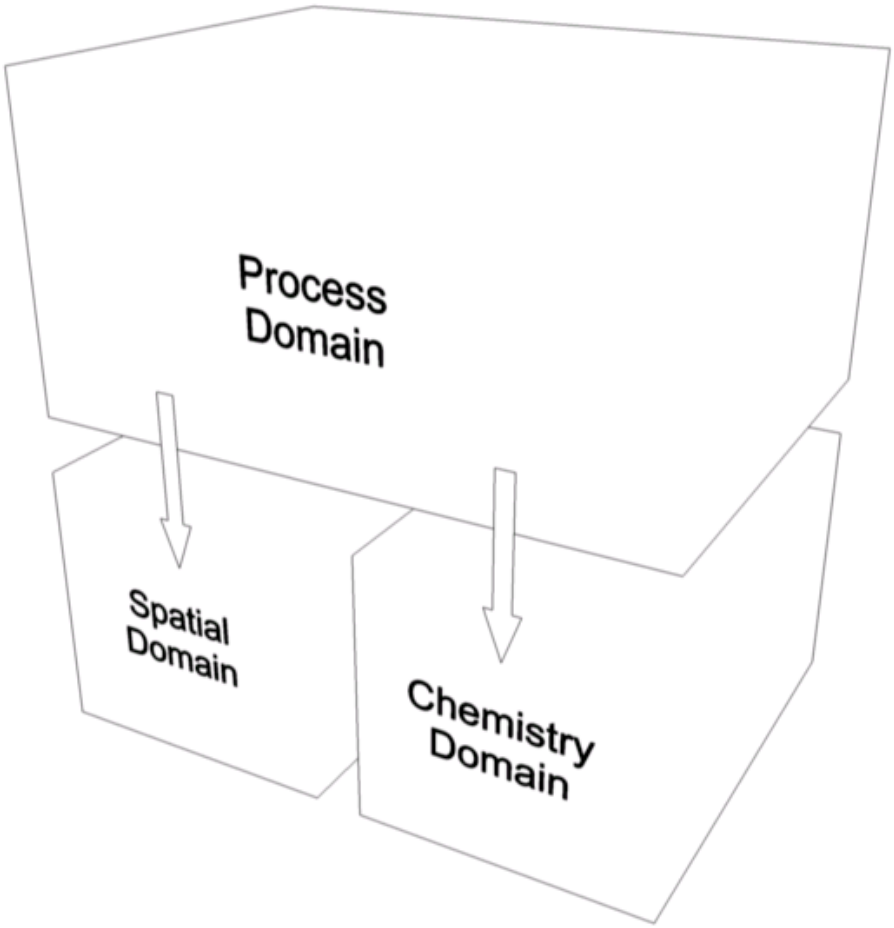


Figure 6.2: Depiction of the high-level dependencies amongst the domain ontologies for process-engineering, spatial-modelling, and chemistry.

6.2 Summary of the Chemistry Ontology

The chemistry ontology allows the practitioner to assemble run-time descriptions of chemistry-related abstractions, such as chemicals, phases, and reactions. While the choice of the smallest unit of matter is arbitrary—depending on the particular discipline—the present study considers "element" to be the lowest layer concept type.

Each participant entity required some means of unique identification, to make the assembly of descriptions of abstractions dependent on these entities in themselves unique. The identity of each atom was characterised according to its related element. The identity of each chemical was resolved through the related molecular structure, hence requiring the atom and bond abstractions.

The present study had little need for specifying molecular geometry; however, such information does become useful for uniquely identifying different chemicals, rather than using less unique means, such as alphanumeric naming schemes. In light of this need, identity is resolved purely through in-memory uniqueness of the run-time instances of these concept types.

The identity of each phase might have been characterised by the constituent chemicals; however, distinctly different phases can have the same component chemicals. In some cases, the difference can be characterised by the numerical fractions of each chemical in the phase. Unfortunately, pure substances relegate this approach. Additionally, as mentioned earlier, we are in the initial business of describing the participant phases involved in the process prior to actual numerical modelling of the accumulation of these phases. The question of qualitative identity for each phase is still possible, but becomes implementation dependent—meaning independent of any universally identifiable standard allowing characterisation, and hence hidden in a manner that is specific to the particular simulation implementation.

There was also the need for describing relevant reactions, and hence also the thermodynamic properties of the constituent chemicals and phases. As will be discussed shortly, information for describing a reaction within the field of chemistry falls slightly short of that needed within process engineering, where one must be able to associate reactions to regions, boundaries, and surfaces—in other words, through specifying more information.

Apart from characterising the identity of these chemistry-related entities, there was also a need to be able to assemble these building blocks into higher-level abstractions—such as assembling descriptions of molecules, phases, and reactions. In addition, certain abstractions had to be assembled without violating certain rules—for example, that the stoichiometry of chemical reactions be balanced.

Lastly, thermodynamic properties had to be associated to each chemistry-related conceptual entity for access during simulation. Related to this problem is the need for specifying the equations that described each thermodynamics property, remembering the earlier restriction that we wish the practitioner to never write code.

6.3 Process-Engineering Ontology

As mentioned earlier, process modelling attempts to reproduce the behaviour of processes through assumption and conceptual representation. The construction of this representation is constrained, according to the rules of assembly for the related modelling abstractions. These rules are independent of the mathematical representation, with multiple modelling approaches available to represent each concept type. The descriptive assembly of abstractions can be built prior to selecting mathematical models.

By observing the assembly sequence for abstractions in the case study, we can obtain the dependency order for these abstractions in the related layered architecture, and hence the ordering of these layers from lowest-to-highest ranking. The ordering is obtained by following a simple rule of data modelling: minimise the number of cyclic dependencies by disallowing lower-layer abstractions from depending on higher-layer abstractions. In other words, base the abstraction order on a directed, acyclic graph of relationships. Using this heuristic, the present section identifies the lowest-to-highest layer ordering. These layer types are now presented.

6.3.1 Particulate Layer and System Layer

The lowest layer of a process description involves specification of materials that arise throughout the process, constructed from the concept types of chemistry. Many chemistry abstractions exist independently of the instances of matter and energy occurring within space and time. What does tie these abstractions to processes is their association with systems and streams, entities that have location, even if that location is conceptual, and not yet spatially defined—although this may change as a model evolves.

It would appear that the "system" is the most appropriate candidate for the lowest-layer abstraction. However, another abstraction arising within process simulation prevents this arrangement—number-averaged materials, such as populations of particles, bubbles, droplets, and porous space, collectively termed hereafter as particulates. The key problem relates to identity: the conceptual process description cannot use numerical quantities to identify each particulate. The description must be assembled from particulate *types*, such as exemplified by the statement "the streams and vessels found in the process contained silica particles". We thus collectively refer to the "silica particles" as a descriptive type to be added to the systems and streams involving that material.

Each particulate must be identified in some manner, such that subsequent number balances for each particulate only apply to accumulations and flows of the same particulate. Hence, the lowest abstraction layer is the *particulate layer*, within which each unique particulate type can be specified for use in assembling descriptions of the systems and streams within the process. Above this particulate layer lies the *system layer*.

Particulates and systems should be considered multi-phase composites. The phases within a system should additionally be able to encapsulate multiple particulates. For example, a vessel's liquid contents might simultaneously contain multiple phases, with different particulates suspended in each phase. Thus, we should be allowed to associate a set of phases to each particulate type, a set of phases to each system, and a set of particulate types suspended within a given phase encapsulated within the system. In this way we assemble the conceptual description of the materials composing the overall process.

Often the initial choice of a system involves no specification of the related location or geometry. Only during model development might a spatial definition be necessary, to subsequently dependent phenomena. Additionally, spatial definition progress at different paces for each system within the simulation. The initial model ignored a spatial description for the systems involved; the final model required the spatial definition for some systems, while others remained spatially unspecified. Hence, not only must the ontology allow materials and particulates to be tied to systems, it must also support the ability to optionally associate each system with a spatial region, and to have systems at different degrees of spatial completeness.

For an example of spatial completeness, consider that a radiative heat-loss model for inclusion in a simulation require that the geometry for participating systems be defined. Consider also that due to the evolutionary nature of model selection, that certain defined systems in the simulation are not yet added to this radiative model, or that they are never to participate in the radiative model. The geometry of these excluded systems is either yet to be chosen or never to be chosen. Hence the overall spatial representation would involve a mix of systems that were either (a) spatially defined or (b) located conceptually without any spatial representation.

It would also be feasible to allow specification of geometry for each particulate type. Note that, as will be discussed shortly, particulates are abstractions that exist within a separate frame of reference to that of systems: the specified geometry is an average of all particles belonging to that particulate. It does not make sense to include particulate geometry within the same spatial reference used to orient systems. It follows that the particulate geometry does not participate in the same spatial representation of section 6.3.2.

The above intersection between geometry and materials introduces constraints on the interactions between system and particulate abstractions. The primary constraint relates to their relative frames of reference and the degree of detail to be included by the practitioner. The boundaries of a system—conceptual or spatial—introduce an abstraction horizon similar to the event horizon of a black hole: we can conceptualise the entities enshrouded by the system boundaries, but not explicitly locate any internal structure except through approximation. Put another way, a particulate only has meaning within the boundaries of a system. The internal structure of the system becomes a conceptual representation relative to the frame of reference for a system.

The goal of having the particulate abstraction is to significantly reduce the amount of information involved in tracking the location and orientation of every particle belonging to an instance of a particulate type. However, there should be no constraint preventing a practitioner from promoting an instance of a particulate type from within a system, making every member particle its own system existing on the same plane of reference as the parent system. The system that previously bounded the related particulate instance then becomes a separate system that exists conceptually alongside every particle system. It follows that we could then place particulate types into the newly promoted particle systems.

Another characteristic, entertained by many practitioners, is the ability to encapsulate systems within systems ad infinitum. The rationale to include this facility requires clear delineation concerning its exact intent. From a practitioner's perspective, the ability to encapsulate systems may imply (a) that the encapsulated system is spatially contained by the parent system, (b) that constraints exist preventing higher-layer abstractions connecting the contained systems and systems outside the parent system, (c) that repeated encapsulation provides a means of grouping together related systems to understand overall movements of materials and energy throughout a large process or (d) varying combinations of these three intents.

The first intention is redundant because the spatial representation over which the process description is layered already provides this functionality. Any duplication of purpose would generate confusion for the practitioner and create conflicts should system encapsulation and the spatial representation contain opposing conditions.

The second point provides a more credible intent, but is rendered moot due to an overlap of purpose with (a) the boundary abstraction introduced in the next section, (b) the particulate abstraction and (c) the spatial representation. The particulate abstraction provides an entity located conceptually, and hence hidden, within the boundaries of the system class. Placing a similar intent upon system encapsulation would create confusion. Additionally, systems possessing a spatial representation already possess constraints on their interactions with other spatial systems, due to the very nature of 3D geometry. As will be seen shortly, the placement of boundaries by the practitioner identifies the interfaces allowing system-system interaction and system-particulate interaction. This approach should provide adequate facility for the practitioner to restrict the set of interfaces across which higher-layer interactions may be bridged. For these reasons, the use of system encapsulation to indicate constraints on interaction provides a potential source of confusion. Such an approach sits awkwardly alongside the desire for clean delineation of purpose amongst different abstraction layers.

The final intent, to track bulk movements of energy and materials amongst groups of systems, does provide a useful conceptualisation tool. Under such a framework, a practitioner can overcome the immediate complexity of a massive process model to quickly visualise its intent. This trait provides sufficient reason to incorporate this form of system encapsulation that, for the reasons mentioned so far, should support this intent in its purest form, without any implication of spatial containment or interaction constraints. The direct upshot is that any system encapsulating another system may never possess a spatial representation nor itself immediately contain any material—phase or particulate—other than the encapsulated systems. Hence, recursive system encapsulation descends until we reach the lowest system into which phases and particulates may be added during model assembly.

This arrangement has the implication that encapsulating systems cannot participate in interactions involving higher-level abstractions, both due to the restriction preventing (a) association with a spatial representation and (b) the addition of phases or particulates. Process modelling primarily aims to reproduce behaviour arising from known driving mechanisms. Under the arrangement presented above, the ability to recursively encapsulate systems does not address this task, as it

simply groups the accumulations within those systems. We must draw a point somewhere in this recursive descent of systems where we provide a description of the conceptual mechanisms driving the process. This ability proceeds from the innermost encapsulated system, where phase and particulate addition are allowed.

Lastly, as mentioned in section 6.4.3, a means should be provided to define the environment into which process systems are placed. Such an environment would provide an external continuum with which to interact, and should be treated as a special system, possessing no accumulation of mass and energy. The reasoning for this last characteristic is that the environment stretches off to infinity with respect to the spatial representation. Only specification of intrinsic properties—independent of the quantity of material present—for the environment should be allowed: temperature, pressure, composition, and so forth. This special, environment system has ties with particular management abstractions presented shortly.

6.3.2 Stream Layer

The stream abstraction aggregates phases and particulates that move between systems en masse. Potential constraints on stream placement are imposed by choices in descriptive abstractions in higher layers, as made by the practitioner. For example, by approximating the flow patterns within a process vessel's bath by using a 2D cross-sectional model, we would be prevented from placing a stream such that the introduced flow produces axial flow within this spatial system.

The domain ontology should include both discrete and continuous streams. Continuous streams can be considered to switch on and off in addition to having their flow rate modified. The practitioner must also be able to specify if a stream involves instantaneous discrete flow.

Lastly, the ontology must allow the connection of streams between systems while observing that the material constraints of the adjoining systems coincide with those in the stream: the stream should contain a subset or the complete set of phases and particulates in the two systems, as well as the same mathematical representation of the particulate population.

6.3.3 Boundaries Layer and Variation Layer

The case study involved the modelling of various transport phenomena, namely convection, diffusion, conduction, and radiation. Modelling of such phenomena requires specification of both the boundary conditions and spatial variations for the properties that drive the related mechanisms. Boundary conditions need to be associated to boundaries. Support is also required to specify the type of boundary condition, for example Neumann and Dirichlet boundary conditions. In terms of the present work, a boundary is a collection of surfaces across which identical conditions apply. These surfaces need not require explicit spatial representation: transport mechanisms may involve conceptual boundaries.

For example, a model might involve convective mass-transport of a chemical through a liquid phase to the surface of particles suspended in that phase. In this case, the boundary is the fluid-particle interface, which has been treated conceptually through the use of the particulate abstraction, instead of specifying an explicit 3D spatial representation for the location, geometry, and orientation of every suspended particle. These dependencies require boundaries to be associated between (a) adjoining spatial systems, or (b) between the particulate and its immediate encapsulating system. Note that the fluid system encapsulating the suspended particulate might require a spatial representation during the evolution of the remaining models.

As described in section 6.2.2, the practitioner frequently desires to approximate the related transport phenomena according to simplifying geometric properties. Hence the association of variations and boundaries to systems is further restricted by the constrained-continuity rules arising from these simplifications. Only certain approximations of spatial variations can be tagged to particular types of regions: one cannot apply a linear one-dimensional variation to model radial heat transfer through the cylindrical shell of a vessel. Thus, intrinsic support must be provided to prevent inappropriate assignment of variation types to the wrong geometric form.

The need for modelling variation requires practitioner to specify where the related variations are tied and the approximating form of those variations—such as the use of a one-dimensional temperature variation through the walls of a cylindrical vessel. We might also represent the fluid-velocity variation in the axial cross-section of a vessel's contents with a two-dimensional variation.

The specification that a property varies throughout a spatial region does not imply any numerical representation of that variation. While assorted mathematical methods exist to represent variation, a descriptive framework only requires the association of conceptual entities to indicate (a) where the spatial variation arises, (b) the form of variation—such as one-dimensional cylindric-radial, and so forth—(c) the property involved, and (d) the associated boundaries. This conceptual framework ensures valid model assembly and permits the layering of dependent abstractions over these variations. The mathematical partition, presented later, supports the facility to choose the associated mathematical representation to be tied against the variation.

The above requirements place the abstraction layer for boundaries immediately above the system layer. We shall follow the convention of naming this layer the *boundaries layer* to avoid confusion with the term "boundary layer" encountered in the field of computational fluid dynamics. It follows also from the above arguments that the *variation layer* is placed immediately above the boundaries layer.

6.3.4 Mechanism Layer

Many of the abstractions placed in lower layers of the concept stack provide a basis over which we may map driving mechanisms. These latter abstractions include the mass- and heat-transport phenomena of conduction, diffusion, convection, and radiation, as well as chemical reaction. The abstractions reside in the *mechanism layer* located immediately above the variation layer. The various mechanism types may be characterised according to their relationship with regions, boundaries, or both. Additionally, each type of mechanism requires the presence of specific abstractions in lower layers for their own addition to be valid. The following list describes the requirements of each mechanism in more detail:

Conduction

Conduction provides a driving mechanism for heat flow through regions, arising purely from spatial temperature gradients. This abstraction must be associated with a spatial temperature variation. Hence an instance of the conduction type may be tied to a specific system or particulate that possesses a spatial description. Note that only one instance of the conduction type can be associated to a particular region.

Diffusion

Diffusion provides a driving mechanism for chemical flow at the molecular level, arising purely from spatial concentration gradients. Diffusion mirrors the characteristics of conduction, except that the practitioner must also specify the chemical species involved. This abstraction must be associated with a spatial variation for the chemical-species concentration property. Hence an instance of the diffusion type may be tied to a specific system or particulate that possesses a spatial description. Unlike conduction, multiple instances of the diffusion type can be associated to a region, one for each chemical species involved. The upper number of instances is constrained for a given region according to one less than the number of chemical species present in the particular region.

Convection

Convection provides a driving mechanism for mass- and heat-transport phenomena, arising from fluid flow created by (a) temperature and concentration gradients and/or (b) imposed mechanical mixing. The creation of an instance of a convection type implies the presence of related fluid flow for the spatial region of interest, and hence the presence of a bulk fluid state. Spatial variations for temperature and concentration might also be specified should they vary excessively throughout the spatial region. Like diffusion, mass-transport convection requires specification of the participating chemical species, and may be associated multiple times for a region based on the number-of-species constraint described above for diffusion. Because convection may involve coupled mass- and heat-transfer behaviour, an instance of the convection type should handle all information for all instances of the convection behaviour of both mass- and heat-transport phenomena.

Radiation

Radiation provides heat transfer arising from electromagnetic radiation exchanged amongst surfaces and volumes of abstractions possessing spatial properties. The mechanism layer must provide the facility to allow practitioners to identify the boundaries and regions participating in this mode of heat transfer. Because radiative exchange is allowed for boundaries and regions, both spatial systems and particulates may be associated with an instance of the radiation type.

Chemical Reaction

The chemistry domain previously described a reaction according to its participants, each participant identified by the related chemical and parent phase. The process-engineering domain extends this notion by requiring the additional specification of the parent system or particulate. This arrangement supports the association of a chemical reaction (a) with a given system or particulate, and (b) with a given system-system boundary or system-particulate boundary. A practitioner must also be able to identify each reaction as forward or reversible, and potentially in equilibrium or rate controlled.

6.4 Summary

This chapter has identified the concepts specific to the process-engineering domain. The work has defined the layer order, layer characteristics, and dependencies of the process-engineering domain upon the chemistry domain and spatial domain. In the next chapter, a formal specification is written for the domain, to identify the types required to support these process-engineering concepts. This formal specification is known as the domain ontology. The domain ontology embodies the types, type collaborations, type inheritance hierarchy, layer ordering, and layer characteristics for subsequent interpretation by the architecture—essentially a more strict definition than the discussion presented here.

Chapter 7: Process-Engineering Ontology Specification

We now present a specification of types for the example domain ontology, to match the domain analysed in the previous chapter. The specification presents the types to be provided by the domain ontology, and indicates how these types fit into the architecture specified in Chapter 4. Like that chapter, the discussion is presented in an implementation independent manner, and does not rationalise the inclusion of the various types—which has already been performed in Chapter 6.

7.1 Introduction

The present chapter specifies the process-engineering ontology designed in the previous chapter. The chapter explains how this domain ontology fits into the layered architecture described in Chapters 3 and 4. For clarity, the specification focuses solely upon the domain ontology for process engineering. Mention is made of relations to types in the chemistry and spatial domains. However, because the example aims to (a) show how domain ontologies fit into the architecture, and (b) show how architects go about selecting types for a domain ontology, the chapter does not provide a specification for these other domains. Types shown from these other domains are used in specifying types in the process-engineering ontology. The author stresses that the chapter makes no attempt at defining the complete set of types for these other domains, nor their type collaborations or layer ordering.

Like Chapter 4, the specification adheres to UML 1.4 notation, viewed from the specification perspective. The UML diagrams aim to supplement the text rather than mirroring every intention of the author: behaviour of each type is described within the text. The UML notational symbol for a package has been used arbitrarily to represent the boundaries of each domain ontology namespace.

The chapter consists of two parts. In the first part, each section defines a group of types, because types residing in different layers of the domain ontology may belong within the same inheritance tree and within several collaborations. It is thus easier to first define types on a per-collaboration or per-inheritance-tree basis. The discussion explains (a) the relationships amongst types, (b) the relationships to types in sublayers, and (c) the relationships to types in other domains. In the second part, each section defines how these types map to individual layers of the domain ontology. The layer types are defined in terms of support for configurations.

Apart from these exceptions, the document applies the same format as Chapter 4 to describe each type. The text draws attention to entity names with italics. For example, *System* and *getPhase* represent entities. The reader should differentiate this approach from that used in the UML diagrams, where italics indicate abstract classes. The type-characteristic keywords—*root, leaf*, *abstract*, and *concrete*—also appear in italics, but should not be confused with entities. Definitions for these four type-keywords may be found in the UML specification (Object Management Group, 2001).

Throughout the domain ontology, readers will note that all base classes are abstract, in accordance with recommended OOD heuristics (Riel, 1996). Only the leaves of the inheritance trees are concrete, and thus instantiable. The domain ontology is meant to be complete for instantiation by the practitioner, rather than extensible: a domain ontology is statically defined to enforce a fixed set of usable types, available to a practitioner at run-time.

Some choices within the architecture are arbitrary and might be argued against depending upon ones viewpoint. The architecture does not espouse its particular arrangement as the solely correct approach. However, the selection of types, and related collaborations, leans in favour of run-time variability while satisfying the Bayesian layering constraints discussed in Chapter 3. Hence, cyclic dependencies within the related type-association graph have been eliminated. The next section commences discussion by examining the various inheritance and collaboration relationships amongst the types.

7.2 Type Specification

This first part focuses upon inheritance trees and type collaborations that make up the domain ontology. Discussion starts with types from the bottom of the dependency order. Note carefully which of the types are realisations of *Concept*. Not every type is a realisation of *Concept*, although many of the types within the domain ontology represent concepts in the classical sense: *Concept* is an essential type for tying ontology-related types into the layers specified in Chapter 4. The reader should note that only types not composed of yet other types are treated as realisations of *Concept*, because these provide the roots for the architecture-related layers to point into the ontology-related type hierarchy. In many cases, the root of inheritance hierarchies are realisations of *Concept*: the act of inheriting from such root types makes the derived types effective realisations of *Concept*.

Furthermore, readers will note that only leaves of the related inheritance hierarchies are concrete—instances of the related types may be created at run-time. Implicit to this statement is that all base classes are abstract. Apart from being a sound OOP design heuristic (Riel, 1996), this approach is intrinsically mandated by the requirements discussed in previous chapters: a given domain ontology is meant to be a complete set of types for use at runtime by practitioners who create instances of the types in memory within a running application. Because inheritance cannot be used by the practitioner to specify their problem—inherent to this arrangement—all leaves must be instantiable. Architects must keep the above three distinctions in mind when designing new domain ontologies. We now commence the definition of the various types.

7.2.1 The *Material* Inheritance Tree

Material *abstract, root*

Subtypes of *Material* classify entities composed of multiple phases. This composition involves a set of associations to the various *Phase* instances composing that material. *Material* allows the practitioner to group phases together without having to spatially define the interfaces amongst those phases—often unnecessarily excessive information for most simulations.

Particulate *concrete, leaf*

A subtype of *Material*, *Particulate* classifies multi-phase populations of particles, whose properties are treated collectively across that population, reducing the overhead of tracking the properties of individual particles. The emphasis is on suspension of the particles within a *Medium*. Models of these materials relate their average number per unit volume of suspending *Medium* to one or more properties characterising the particulate—size, geometry, composition, and so forth. *Particulate* includes rigid particle populations, bubble swarms and droplet swarms.

Medium *concrete, leaf*

A subtype of *Material*, *Medium* consists of multiple phases into which multiple particulates may be suspended. The *Particulate* instances are assumed to by uniformly dispersed throughout the medium. *Medium* instances do not have their own layer, because each instance of *Medium* is unique to the parent *System* or *Environment* instance. This arrangement provides a level of abstraction that can be escalated, if so desired, by instead creating one medium for each phase and then specifying the spatial interface amongst these mediums within the structure of a system. The medium shields any external mechanism-related interactions with the suspended particulates, other than through streams. Mechanisms are allowed to proceed between the medium and suspended particulate.

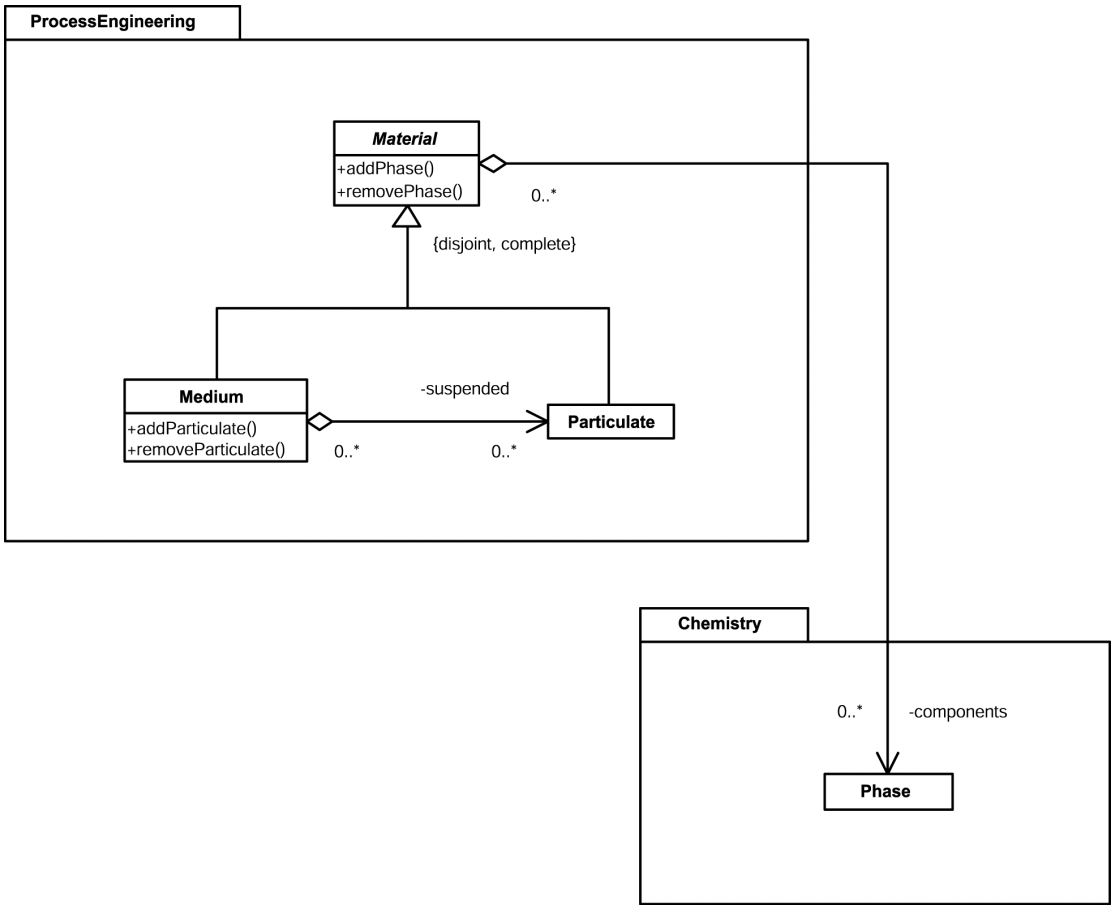


Figure 7.1: Diagram depicting the *Material* inheritance tree and related collaborations.

7.2.2 The *Composite* Inheritance Tree

Composite *abstract, root*

Subtypes of *Composite* classify entities consisting of multiple materials. A *Composite* possesses a *Structure* instance, which owns one or more instances of *Medium* based on the averaged spatial representation chosen by the practitioner.

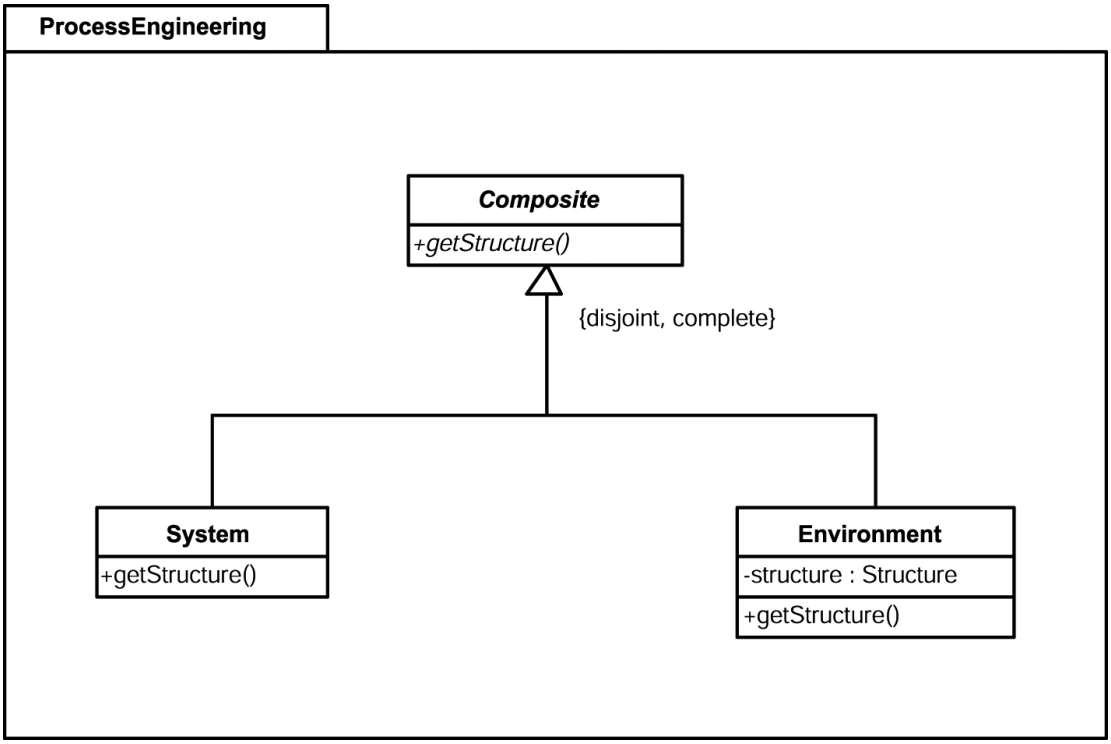


Figure 7.2: Class diagram depicting the *Composite* inheritance tree.

Structure *concrete, leaf*

Structure classifies the multi-material that uniformly permeates the interior of a composite. A structure consists of one or more mediums, each separated by spatial interfaces defined by the practitioner. Hence, a structure consisting of only one medium needs no definition of spatial interfaces: the medium solely permeates the interior of the composite.

As the interior of a composite is uniform, a *Structure* provides an average of the internal geometric structure, rather than an explicit 3D specification for the medium-medium interface throughout the entire volume. A structure averages the internals of a composite on a per-unit-volume basis, and is defined by the practitioner on a unit-volume of space. No definition is made as to whether each medium is rigid or fluid. Note that the scale of the unit volume is relative: the practitioner may choose a structure on a microscopic or macroscopic scale for their particular problem.

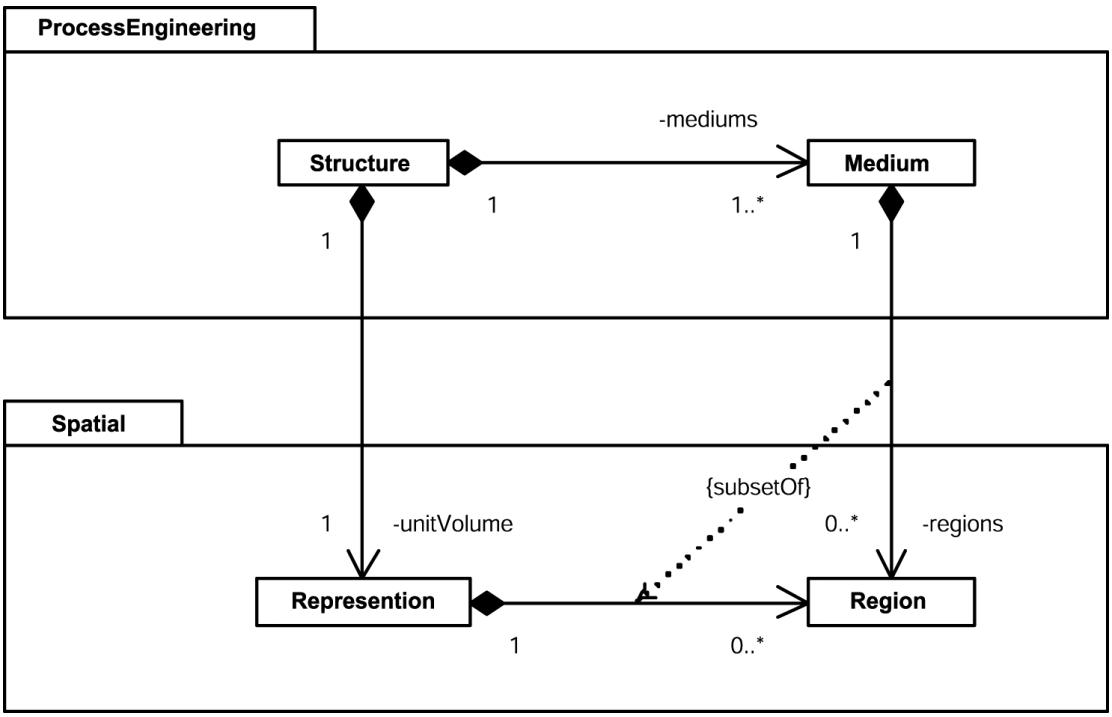


Figure 7.3: Class diagram showing the collaboration and constraints for the Structure type.

Environment *concrete, leaf*

A subtype of *Composite*, *Environment* represents the continuum that surrounds all systems within a *Locale* of a simulation. The difference between *Environment* and *System* is that *Environment* has no bounding volume, although it shares borders with systems placed within that *Environment*.

System *concrete, leaf*

System classifies a multi-material *Composite* bound by a control volume. Only the boundaries of this control volume have location, be that in a conceptual or spatial sense. *System* instances may be positioned within a spatial representation through association to a region in a spatial representation. Systems may encapsulate other systems, provided the encapsulating system does not possess any immediate materials or associates to a region in the spatial representation.

The boundaries of *System* hide the explicit location of their contents: the *Material* type defined earlier has meaning only within the boundaries of a *System*, because *System* exists on a conceptual tier above *Material*. This approach allows the practitioner to treat a medium or particulate as a system, allowing explicit specification of the geometry and location of the material boundaries, rather than spatially averaging the material geometry. In such circumstances, the entity is no longer a *Particulate* or *Medium*, but rather a *System* with a spatial representation. The system or environment that originally contained the material then shares borders with the new system, rather than being conceptually encapsulated by the system.

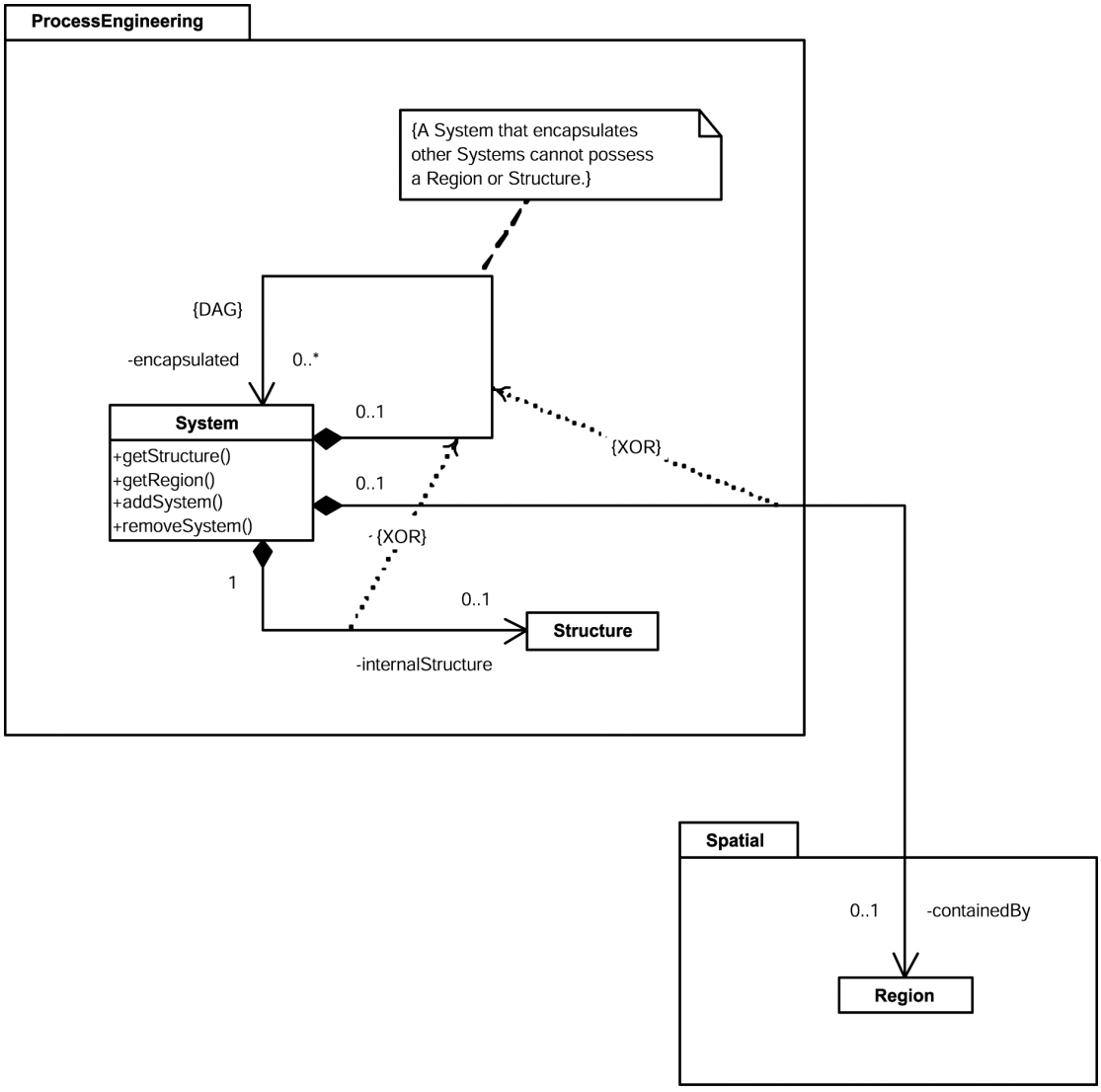


Figure 7.4: Class diagram depicting the constraints and collaborations for the *System* type.

A system for which a spatial boundary has not been specified is termed a conceptual system. This arrangement conveniently allows the practitioner to create systems for which explicit knowledge regarding the related spatial boundaries are not yet specified or are never to be specified. Once a system has been associated a region, that system is then a spatial system. Only one region may be associated to system, locked in at assembly time—the region cannot be exchanged or removed at solution time. Systems may encapsulate other systems provided the encapsulating System does not have materials specified as its immediate contents—other than the encapsulated systems.

7.2.3 *Stream* Related Collaborations

Stream *abstract, leaf*

Stream classifies the macroscopic movement of material to, from, or between systems. A stream consists of multiple phases containing uniformly suspended particulates, much like a *Medium*. The stream must connect to a *Medium* in each *System*, the constraint being that the constituent *Phase* and *Particulate* instances in the stream are a subset of the connected medium(s). The practitioner may perform the following assembly-time modifications to the connection and transience qualities of a *Stream* instance.

A stream may connect two systems, or flow into or out of a single system. A source stream connects with one system, into which the stream flows. A sink stream connects with one system, out of which the stream flows. A link stream connects two different systems, allowing flows from one to the other in a directed manner. A stream may either flow continuously over time (including zero flow) or flow as instant, discrete quantities. Subtypes of *Stream* classify this flow-type property.

DiscreteStream *concrete, leaf*

A discrete stream flows in discrete, instant quantities.

ContinuousStream *concrete, leaf*

A continuous stream flows continuously over time, including the possibility zero flow.

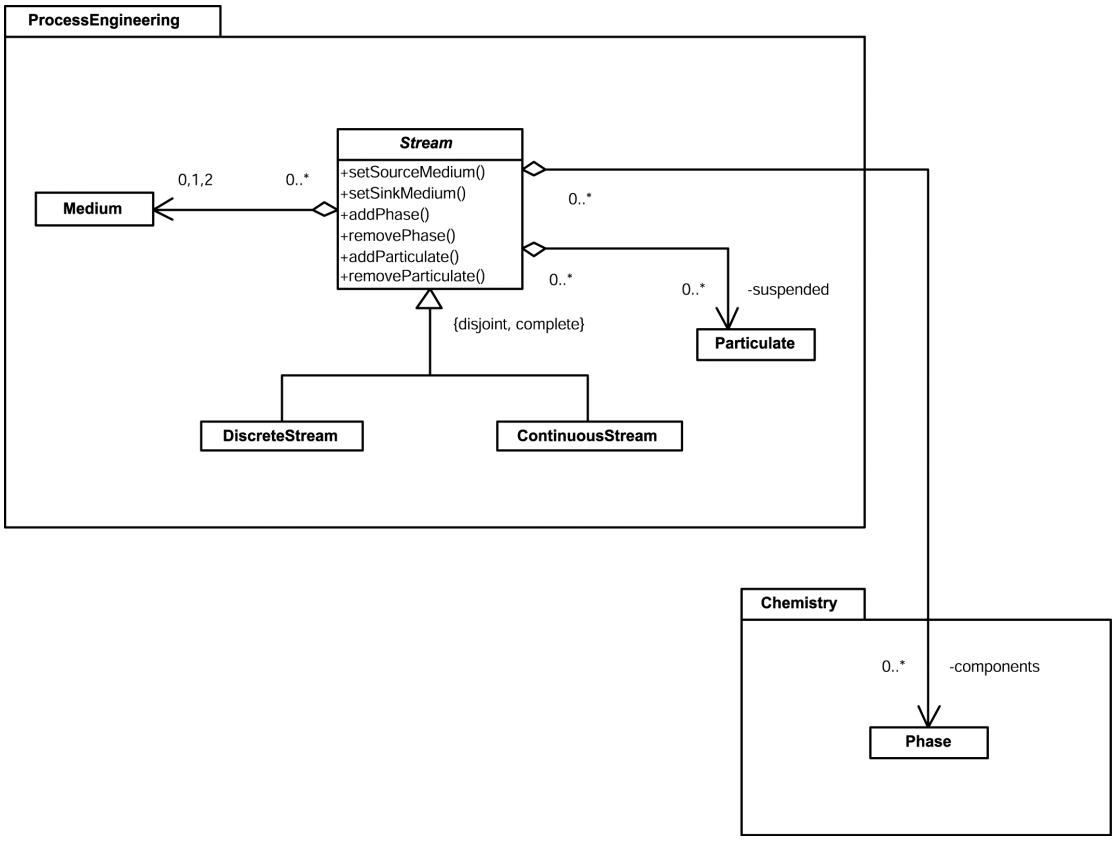


Figure 7.5: Class diagram depicting the behaviour of *Stream*.

7.2.4 *Boundary* and Related Collaborations

Boundary *abstract, root*

Boundary represents any boundary between two materials where identical conditions are met—for example, constant temperature or heat flux. A boundary thus differs to a spatial surface. Boundaries may be specified (a) between two systems, by targeting the *Medium* of each *System*, (b) within a single *System* between the related *Medium* and an encapsulated *StructuredMaterial*, and (c) as a spatial boundary between two spatially-defined *System*s.

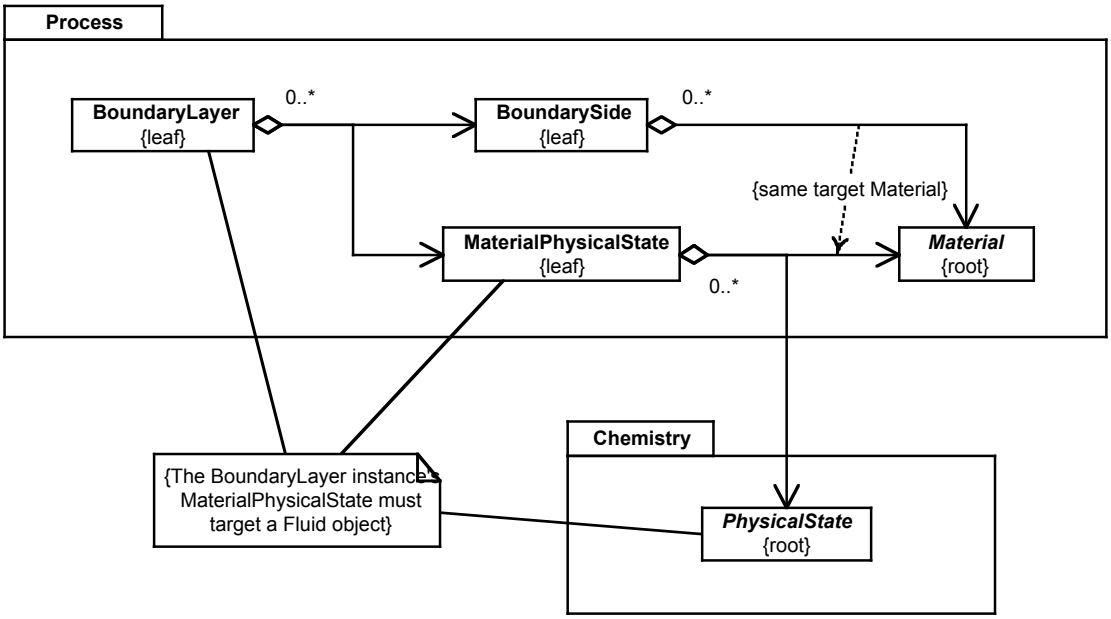


Figure 7.6: Class diagram depicting the collaboration amongst *BoundaryLayer*, *BoundarySide*, *MaterialPhysicalState*, *Material*, and *Chemistry::PhysicalState*.

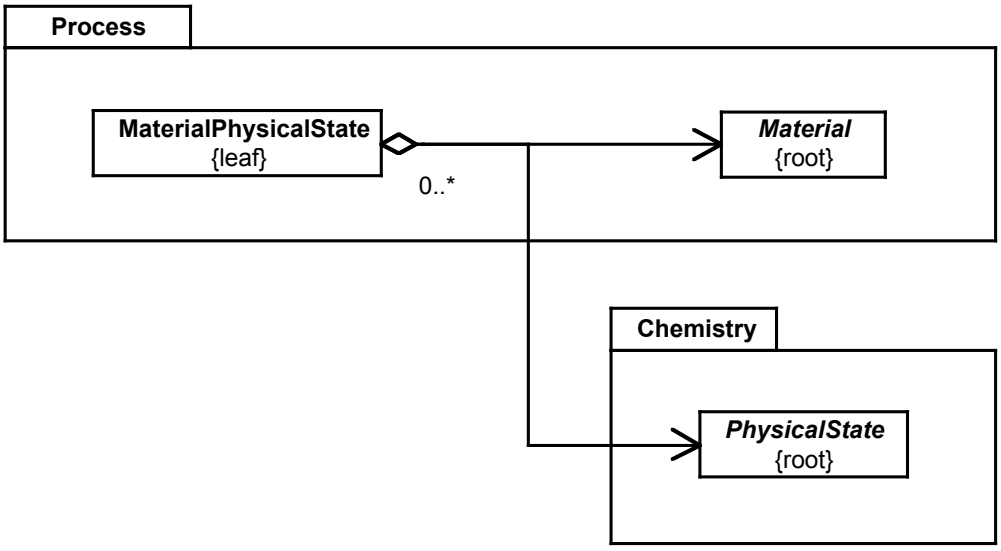


Figure 7.7: Class diagram depicting the collaborations amongst *MaterialPhysicalState* upon *Material* and *Chemistry::PhysicalState*.

7.2.5 *Variation* and the Related *VariationType* Inheritance Tree

Variation *concrete, leaf*

Instances of the *Variation* type represent occurrences of spatial variation. *Variation* provides no numerical representation, only a specification that some variation exists within a particular spatial system. Mathematical approaches for representing variations are specified in the modelling stack.

VariationType *abstract, root*

VariationType collaborates with *Variation* to provide a state pattern for changing the type of an existing *Variation* at simulation-assembly time. Only certain variation types can be assigned to a region of a given geometry. Subtypes of *VariationType* constrain this assignment by querying geometry properties possessed by the target region.

OneDimensionalVariationType *abstract*

A one-dimensional variation involves a property that varies along one spatial axis. This variation need not be rectilinear: cylindric-radial or spheric-radial variations are both one-dimensional variations.

TwoDimensionalVariationType *abstract*

A two-dimensional variation involves a property that varies simultaneously in two spatial axes. Subtypes apply constraints in relation to the geometry of the regions to which it may be applied.

ThreeDimensionalVariationType *concrete, leaf*

A three-dimensional variation involves a property that varies simultaneously in three spatial axes. As a three-dimensional variation can be applied to any form of geometry, there are no constraints of the type of spatial region to which it may be applied. Hence *ThreeDimensionalVariationType* provides a concrete class which need not be subclassed: there are no constraints to be imposed depending on the geometry of the related spatial region.

LinearVariationType *concrete, leaf*

A rectilinear one-dimensional type of variation.

RadialVariationType *abstract*

A one-dimensional variation type extending radially within cylindric or spheric region.

CylindricRadialVariationType *concrete, leaf*

A one-dimensional variation type extending radially within a cylindric region.

SphericRadialVariationType *concrete, leaf*

A one-dimensional variation type extending radially within a spheric region.

7.2.6 The *Mechanism* Inheritance Tree and Related Collaborations

Mechanism *abstract ,root*

This type represents phenomena that drive changes in the states of systems. Subtypes of Mechanism provide the various forms of phenomena.

TransferMechanism *abstract*

This subtype of *Mechanism* provides a base type for phenomena related to heat transfer, mass transfer, and/or the simultaneous combination of both.

Diffusion *concrete, leaf*

Diffusion classifies chemical dispersion throughout fluids as a result of purely Brownian motion.

Convection *concrete, leaf*

Convection represents heat transfer, mass transfer, or mixed heat-and-mass transfer, as a result of fluid motion. The practitioner must specify which transfer type at runtime.

Conduction *concrete, leaf*

Conduction represents heat transfer resulting purely from temperature gradients throughout a System. This type must be applied to regions where a *Variation* exists in the *Temperature* dimensional property.

Radiation *concrete, leaf*

Radiation classifies heat transfer resulting from electromagnetic waves exchanged amongst surfaces of different temperatures. By associating instances of *Radiation* to regions and surfaces, the practitioner indicates which entities in a radiative scene participate in heat transfer through this mechanism.

TransferType *abstract, root*

TransferType collaborates with *TransferMechanism* to provide a Type-Object pattern for changing the type of *Mechanism* during simulation assembly. The *TransferType* holds additional information relevant to the type of *TransferMechanism*. Only the *Convection* mechanism allows the practitioner to specify this type. The other mechanisms automatically default to their respective transfer type to which they are inherently bound—for example, *Conduction* can only be a heat-transfer mechanism.

MassTransfer *concrete, leaf*

A *TransferMechanism* that references *MassTransfer* is itself a mass-transfer related mechanism. The *MassTransfer* instance holds an association to the *Component* instance involved transfer.

HeatTransfer *concrete, leaf*

A *TransferMechanism* that references *HeatTransfer* is itself a heat-transfer related mechanism, involved in the movement of thermal energy.

Reaction *abstract*

A subtype of *Mechanism*, *Reaction* represents chemical reactions amongst chemicals located in phases of materials within systems. Subtypes of *Reaction* relate information regarding the location of participating chemicals.

ReactionParticipant *concrete, leaf*

This class collaborates with *Reaction* to allow the practitioner to specify which chemicals participate in a given reaction. The concept of a chemical reaction in chemistry is expanded upon in process engineering, because the identity of participating chemicals requires more information to be specified by the practitioner. The identity of a chemical in chemistry involves specifying the constituent elements and the parent phase. For process engineering, the practitioner must additionally specify the material to which the chemical belongs.

IntraReaction *concrete, leaf*

An *IntraReaction* involves reaction participants that are all within the one material. For example, a reaction involving only participants from the medium of a system is an *IntraReaction*. The same applies to fiber, matrix, and particulate materials.

BoundaryReaction *concrete, leaf*

A *BoundaryReaction* arises at the *Boundary* between materials. Thus the reaction may occur either between a medium and a constituent structured material within the same system, or between the mediums of two systems. In the case of two spatial systems, the regions of the related systems must share a common spatial boundary.

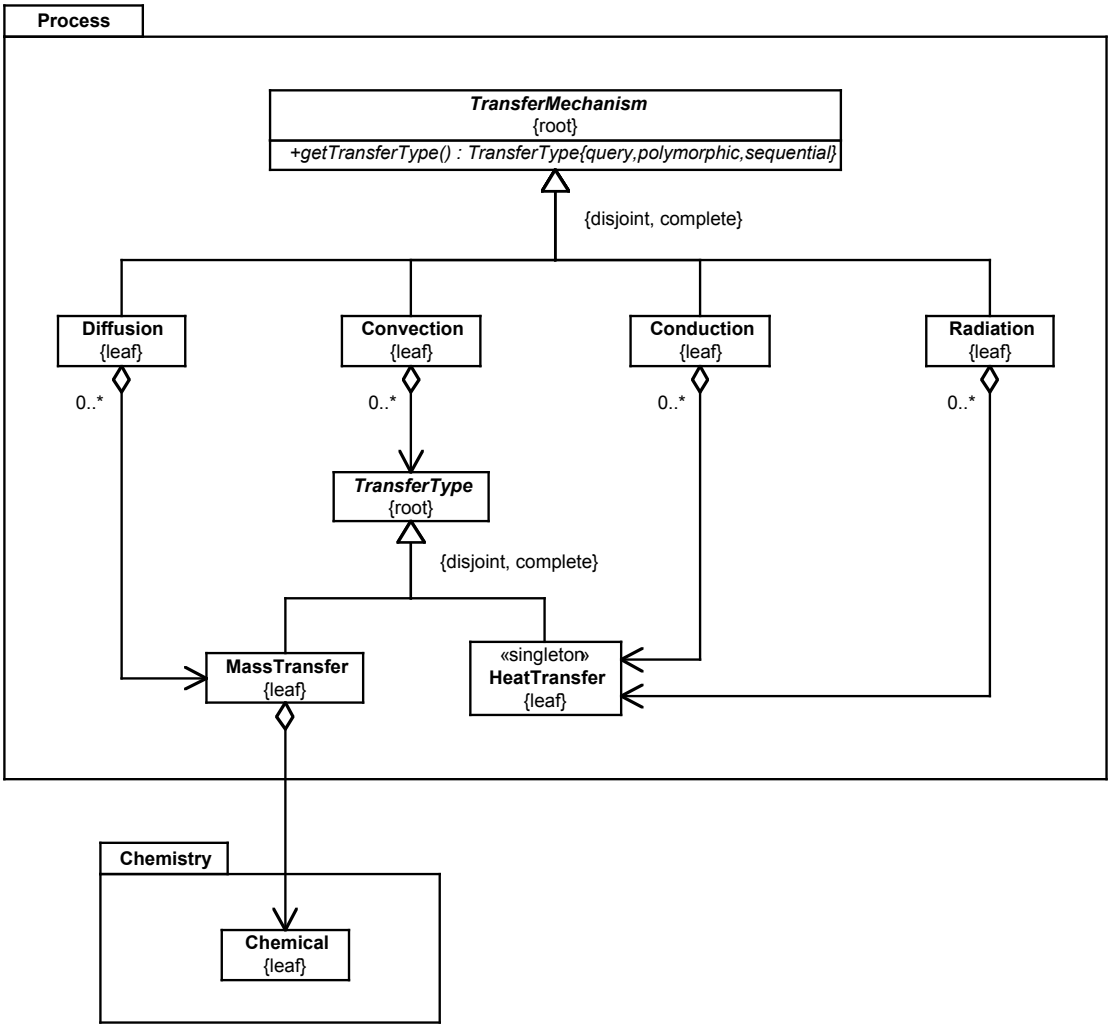


Figure 7.8: Class diagram depicting the relationships amongst the *TransferMechanism* and *TransferType* inheritance trees, and the *Chemistry*::*Chemical* type.

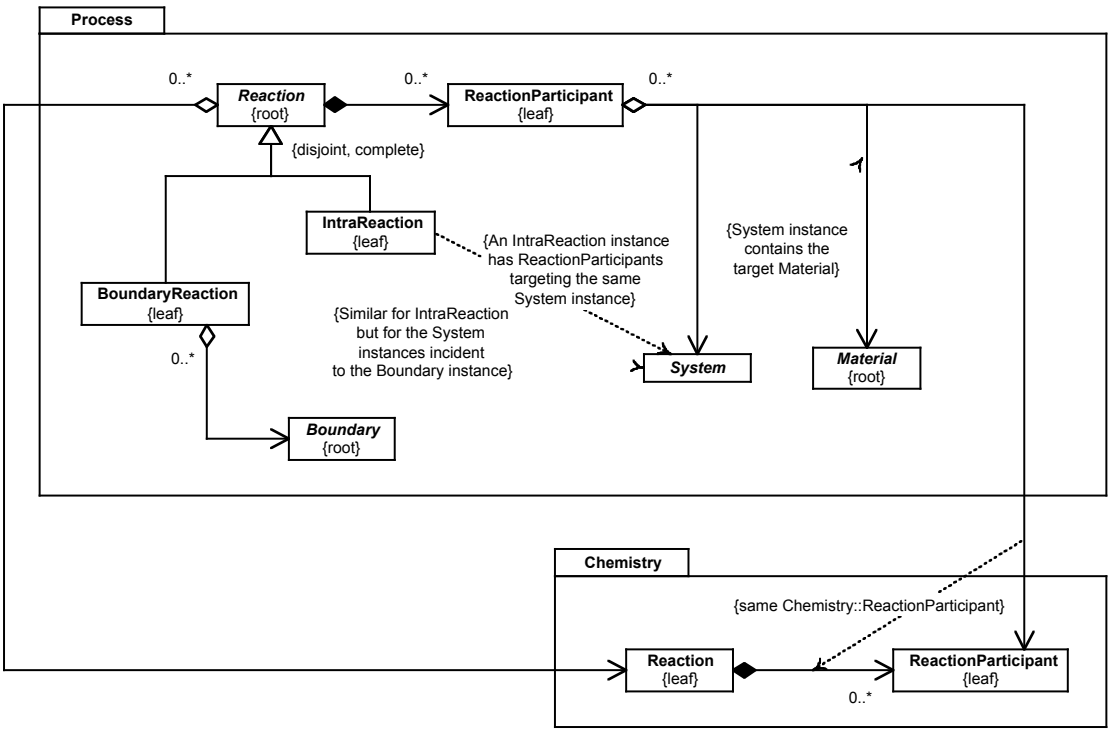


Figure 7.9: Class diagram depicting the dependencies amongst the *Reaction* inheritance tree, *ReactionParticipant*, *Boundary*, *System*, *Material*, *Reaction*, and *Chemistry::ReactionParticipant*.

7.3 Layers

To clarify to the practitioner the manner in which these collaborations are used to assemble conceptual representations, we now tie in selected types to the explicit layers provided by the domain ontology. The trick is to select specific types in the inheritance hierarchies to capitalise on polymorphism to provide the complete range of concept types specific to given layers.

This selection proceeds by keeping in mind that we desire runtime assembly of simulations and that each instance of a concept type is unique. The latter point is important: the nature of reference semantics relies heavily on the uniqueness of instance, and it is these references that allow the practitioner to use the layering in the first case.

Each subsection that follows details the type(s) specific to each layer. Key inter-layer dependencies are highlighted, based on the associations maintained by concepts of higher layers to concepts in lower layers (sublayers). These layer dependencies are what the practitioner must latch onto to provide clarity in using the architecture to map their simulation problems into an assembly of concepts.

Each subsection also explains behaviour related to the layer as a whole, such as to whether configuration are permitted in the layer. Again, keywords defining layer characteristics are presented in italics. These keywords appear on the right side on the same line as the subsection title for the layer. These keywords are as follows:

- *flat* : the layer does not allow the partitioning of its member concepts into configurations nor allows each concept instance to possess different forms. All concept instances created in the layer exist in every configuration in higher layers that allow this facility.

- *configurations* : The layer allows configurations, by which is meant that instances in the layer can be partitioned into configurations whose members are active at the time the related configuration is active. Concept instances are allowed to exist in multiple configurations.

These layer characteristics are mutually exclusive: a layer may possess only one of these qualities. We now list the layers in ascending order.

Particulates Layer *flat*

The lowest layer of the ontology, a practitioner creates instances of *DropletSwarm*, *BubbleSwarm* and *Particles* within this layer. The layer holds references to the base type *Particulate* of these instances. As a flat layer, the layer does not allow the partitioning of *Particulate* instances into configurations: all *Particulate* instances created in the layer exist in every configuration in higher layers allowing this facility.

Composites Layer *flat*

A practitioner creates instances of *System* within this layer. The practitioner can then specify properties regarding each *System*, specifying the phases making up the various medium, specifying the average spatial representation within the *System*, and associating particulates to the medium(s) within the *System*.

Streams Layer *configurations*

A practitioner creates instances of *Stream* within this layer. Following creation the practitioner then adds phases to the contents of the stream—addition is constrained by the phases present in the materials connected to by the stream. This layer is the lowest layer that permits configurations in the domain ontology. Hence, all layers above this layer allow configurations.

Boundaries Layer *configurations*

A practitioner creates instances of *Boundary* subtypes within this layer.

Variations Layer *configurations*

A practitioner creates instances of *Variation* subtypes within this layer.

Mechanisms Layer *configurations*

A practitioner creates instances of *Mechanism* subtypes within this layer.

7.4 Domain Characteristics

The process-engineering domain as a whole requires to be mapped to instances of the spatial domain and chemistry domain. All systems in that process-engineering domain belong to the same spatial representation. Any system mapped to a region must obtain that region from the spatial representation. Additionally, a well-formed model assembled by the practitioner is one in which all regions of the spatial representation is targeted by a system.

7.5 Summary

This chapter has outlined a specification for the types, collaborations, inheritance hierarchies, and layer properties making up the process-engineering ontology. The chapter also specifies the properties of the domain as a whole. The next chapter presents a walkthrough within the practitioner's simulation environment applying this process-engineering ontology and showing how a practitioner expresses a simple simulation problem.

Chapter 8: Demonstration Walkthrough

This chapter demonstrates the use of the implementation through application to an arbitrary process-engineering problem. The resulting walkthrough clarifies how the practitioner's simulation assembly environment is applied in assembling simulations. This chapter presents an initially simple problem upon which complexity is built through two iterations.

8.1 Problem Description (First Iteration)

As the present research originated from the field of chemical engineering, the test problem considered here is also drawn from this field. The initial problem considered is an isothermal, multiphase, molten metallurgical bath undergoing chemical reaction at high temperatures. Excluding bulk material streams flowing through the bath, the system is considered to be independent from its environment. The bath consists of the phases and chemicals listed in Table 8.1; the bulk of the bath consists of the liquid phases uniformly suspending the solid phases. Oxygen gas is continuously injected into the bath while sulphur dioxide gas evolves from the oxidation of sulphur-based compounds present in the bath. The produced sulphur dioxide continuously exits the system as exhaust gas.

The problem considered is a dynamic system: concentrations of chemicals within the liquid phases change over time, due to (a) the gradual removal of sulphur from the bath in the form of sulphur dioxide gas and (b) the accumulation of other oxygen-bound byproducts as iron oxide, magnetite, and copper oxide. Due to the extremely high temperatures and mixing conditions involved, the bath is assumed to exist in chemical equilibrium with the exiting gases. Such an assumption is commonly applied to pyrometallurgical problems of this type. Using this description, an initial simulation is now assembled within the graphical environment.

Table 8.1: Phases and constituent chemicals for the simple problem description.

Phase Name	Phase State	Constituent Chemicals
Matte	Liquid	Cu_2S FeS FeO Fe_3O_4
Slag	Liquid	FeO Fe_3O_4 SiO_2 Cu_2S FeS
Air	Gas	N_2 O_2
7	Gas	N_2 O_2 SO_2 S_2
Silica	Solid	SiO_2

8.2 System Setup

The walkthrough considers simulation assembly from within the simulation-assembly environment. In this walkthrough, a single instance of the ontology repository is provided and already remotely running. Multiple remote running technique repository servers are also provided and preloaded with techniques common to the user's discipline—in this case chemical engineering. These techniques would match the standard techniques that would be published by the community of developers likely to exist in the real-world, online system. The assembly environment is already configured with a downloaded chemical engineering ontology, as well as related chemistry and spatial ontologies. With the environment prepared, the user switches to the assembly perspective within the graphical environment to commence assembly.

8.3 Simulation Assembly (First Iteration)

Based on the problem description, a user of the now-configured graphical environment works as follows. Assembly requires the user to build their representation as instances of concept types depicted in the 3D graphical layers of the application. Assembly proceeds in a bottom-up sequence: concept instances in the bottommost layers are created first, and subsequently, higher-layer concept instances that are dependent on the existence of lower-layer concept instances are created. This procession obeys the rules enforced by the ontology configuration that was originally downloaded.

The user selects the chemistry domain from within the domain perspective; the view transitions into this particular level of conceptual granularity by displaying the layers of this domain. The user navigates to the bottommost layer to begin definition of instances of concepts specific to the problem and which are related to the chemistry aspects of the problem. The bottommost domain relevant to expressing the problem is the chemistry domain. From the domain view presented within the assembly perspective, the user double clicks the chemistry domain node. The view transitions into that domain, presenting its relevant layers. Assembly then proceeds from lower to higher layers, in the sequence of defining chemicals and phases.

The user navigates to the atoms layer to begin description of chemicals found within the problem. The user initially creates the Cu_2S chemical instance: the user creates three atom instances within the atoms layer, associating each with an instance of the related element provided in the underlying elements layer. In this case, one atom instance is associated with the element Copper, with the other atom instance being associated with the element Sulphur. The user then navigates up one layer to the bonds layer. Two bond instances are created, one connecting the Copper atom to one of the Sulphur atoms, the other connecting that same Copper atom to the other Sulphur atom.

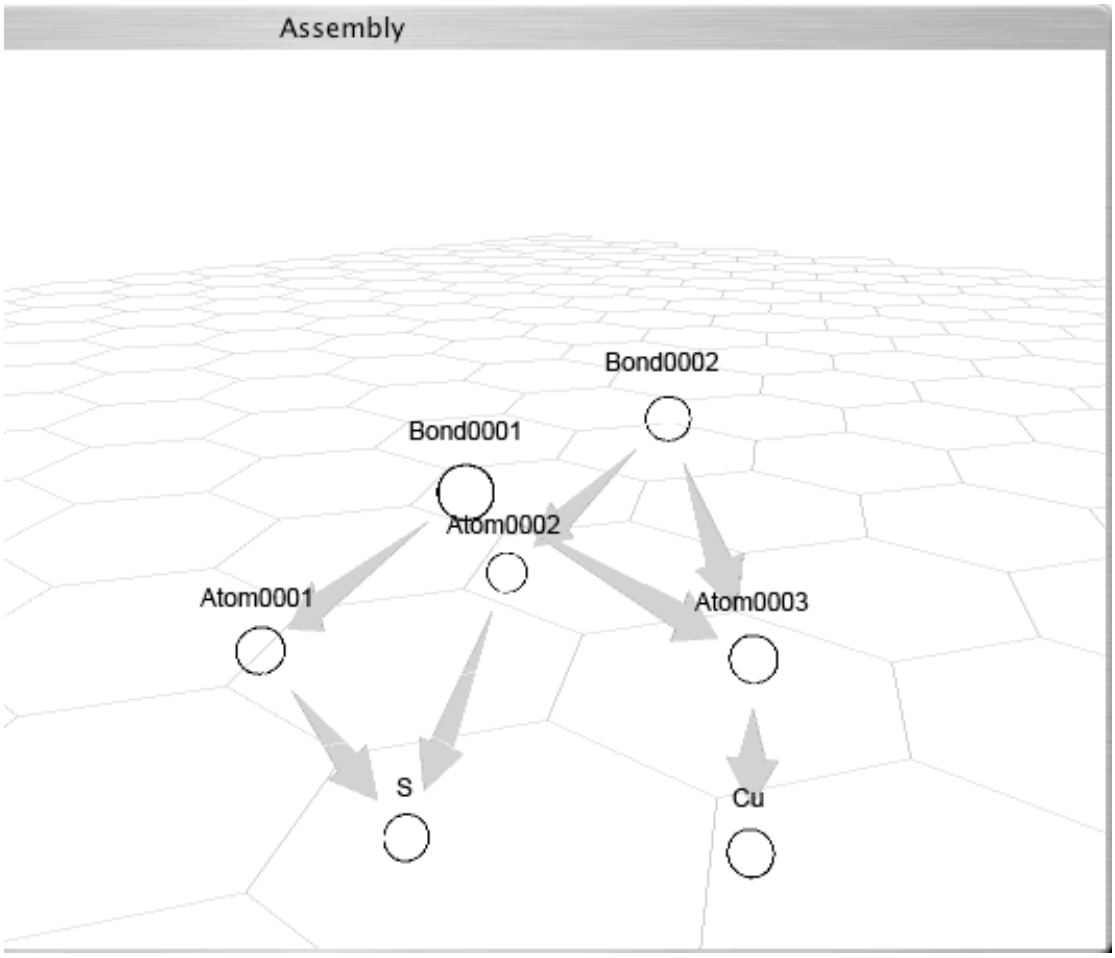


Figure 8.1: Screenshot depicting the bonds layer in constructing the Cu2S chemical.

The Cu_2S chemical instance is then created in the chemicals layer, connecting to the two bonds in the bonds layer and to the three atoms in the atoms layer—depicted in Figure 8.2. This process is repeated for the remaining chemicals listed in Table 8.1—FeS, FeO, Fe_3O_4, Cu_2O, SiO_2, SO_2, S_2, O_2, and N_2.

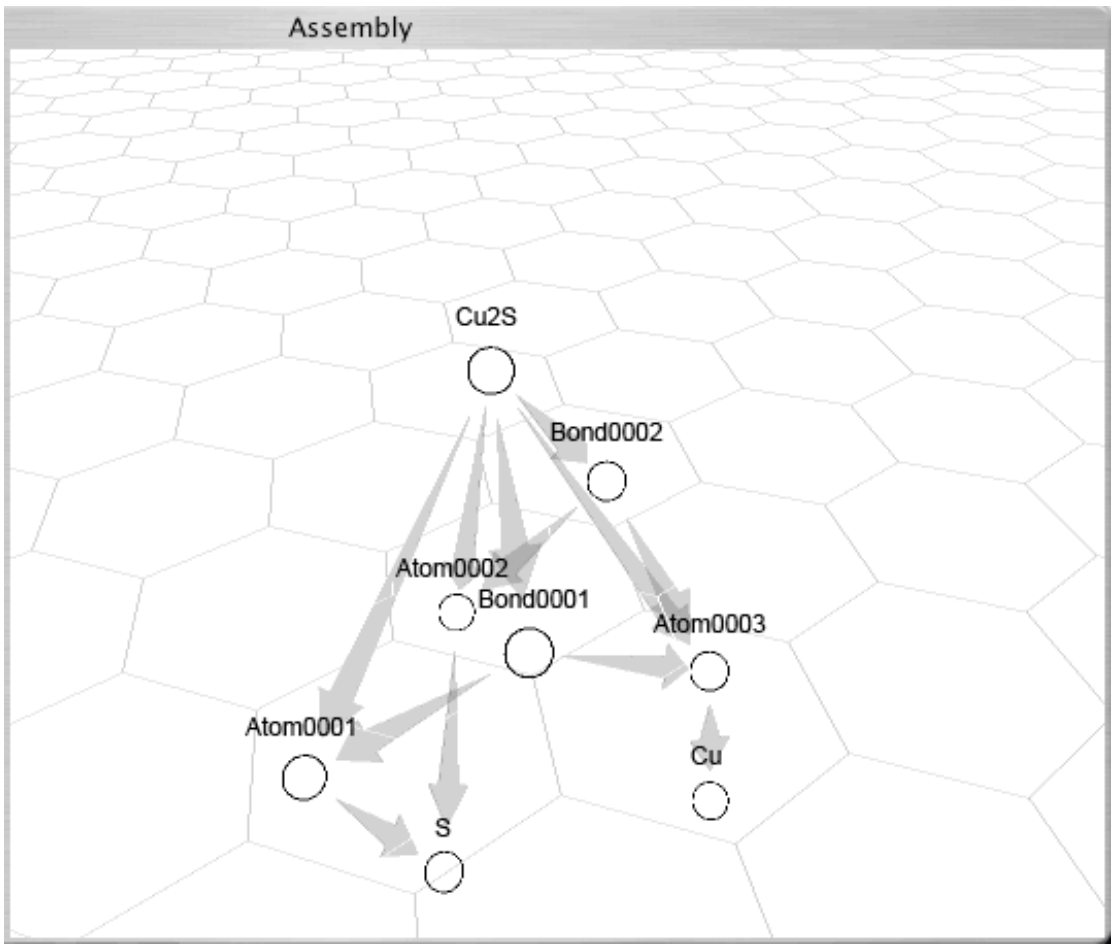


Figure 8.2: Screenshot depicting the chemicals layer in constructing the Cu2S chemical.

With the chemicals defined, the user now navigates to the phases layer. Phases consist of one or more chemicals. For example, the matte phase presented in Table 8.1 is created as follows. The user creates a phase instance within the phases layer, associating this instance to chemical instances found in the chemicals layer. A screenshot of the resulting layer is presented in Figure 8.3. This process is then repeated for the remaining phases presented in Table 8.1.

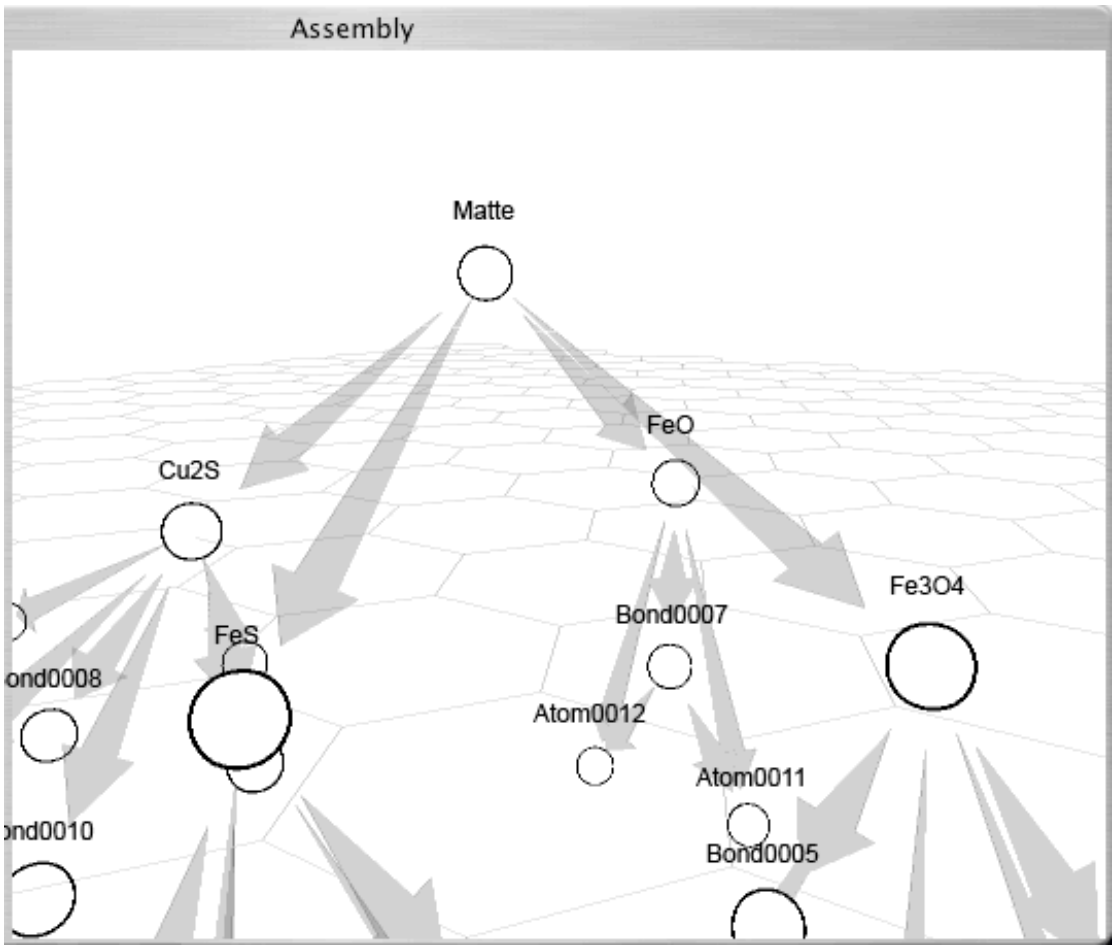


Figure 8.3: Screenshot depicting the Phases layer in constructing the Matte phase.

We next create the relevant systems defining the simple problem. The user now navigates to the chemical engineering domain to commence further problem definition. The user navigates to the system layer and commences creating instances of the relevant systems. The molten metallurgical bath is the only system considered by problem. The user creates an instance of the system concept, associating it with phase instances defined earlier in the chemistry domain. Figure 8.4 depicts the resulting systems layer showing the bath system instance.

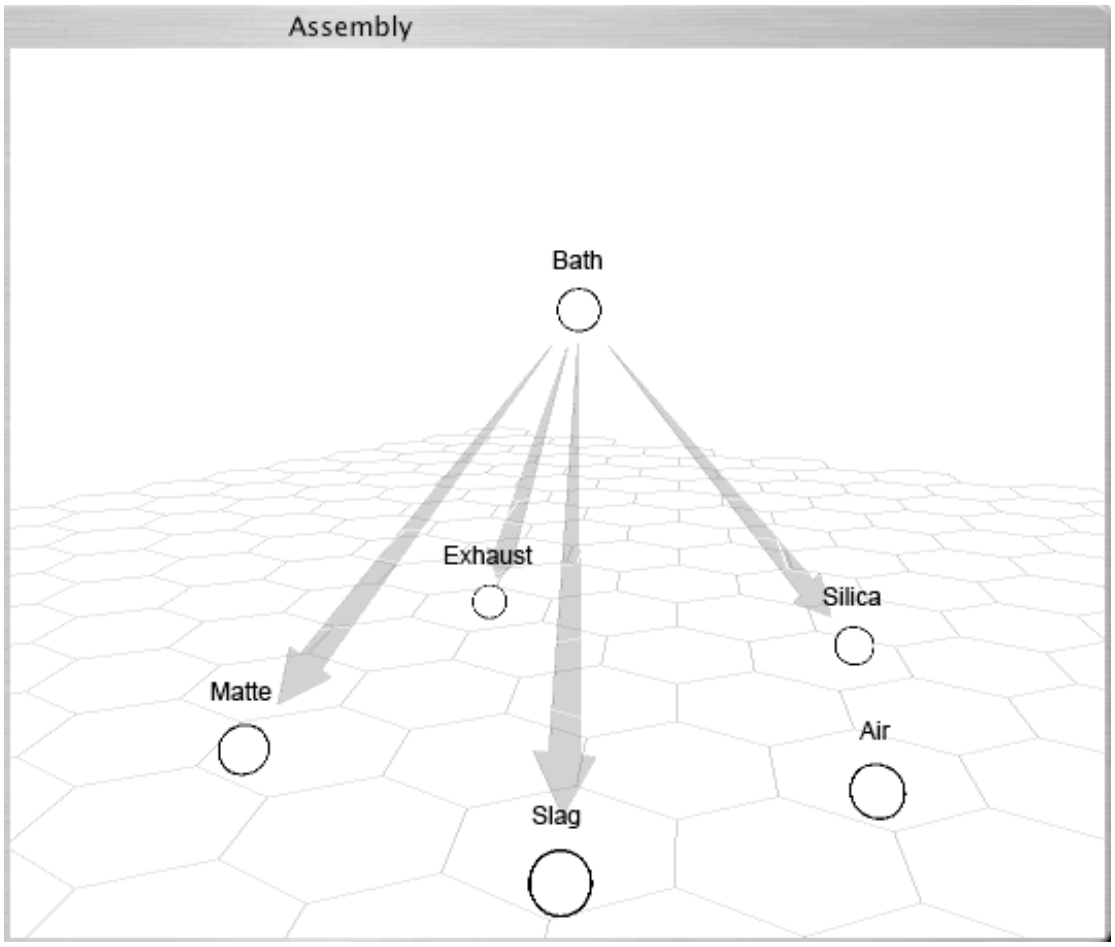


Figure 8.4: Screenshot depicting the systems layer in constructing the bath system.

The user navigates to the streams layer and commences creating instances of the relevant streams. An input continuous-stream concept instance is created for the air stream and associated to the bath system; an output continuous-stream concept instance is created for the exhaust stream and also associated to the bath system. These streams are associated to the relevant consistent phases. Figure 8.5 depicts the resulting streams layer depicting these streams.

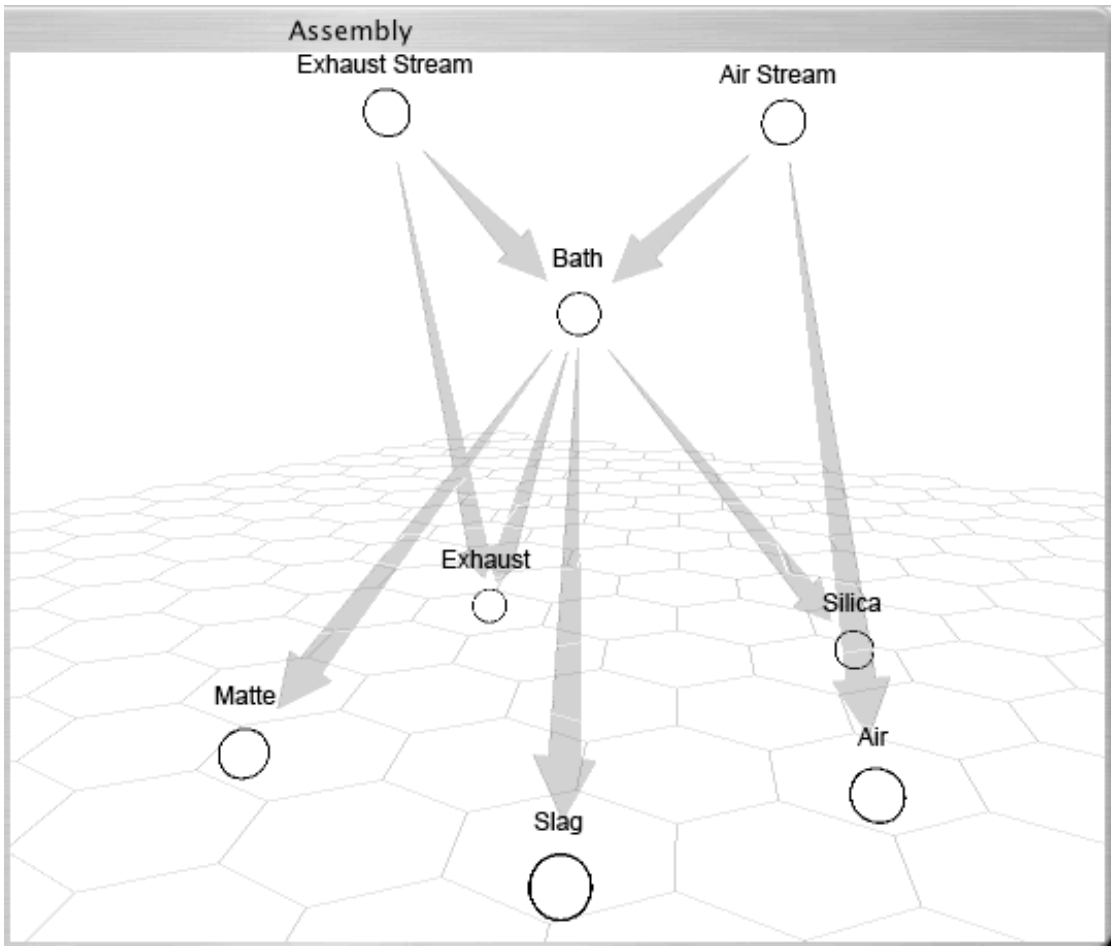


Figure 8.5: Screenshot depicting the streams layer in constructing the air stream and exhaust stream.

8.4 Problem Description (Second Iteration)

The initially simple problem is now elaborated upon. Discrete material streams that flow in and out of the bath are now considered. These streams represent intermittent flows involving near-instantaneous additions of materials to the bath. The range of materials considered are presented in Table 8.2. Furthermore, the streams considered in the simple problem are changed from continuous to semi-continuous streams. Hence, the air stream and exhaust stream are now to be considered as either flowing continuously into the bath, or simply switched off. One more semi-continuous stream is added, in this case, the intermittent addition of silica. An added rule considered in this problem is that the discrete streams described above are only added when these now semi-continuous streams are switched off.

Table 8.2: Phases and constituent chemicals for the simple problem description.

Stream Name	Stream Type	Direction	Phases (see Table 8.1)
Air	Semi-continuous	Input	Air
Exhaust	Semi-continuous	Output	Exhaust
Feedstock	Discrete	Input	Matte
Slag	Discrete	Output	Slag
Flux	Discrete	Input	Silica

8.5 Simulation Assembly (Second Iteration)

Returning to the assembly environment, inclusion of these new complications alters the existing layered representation. The two continuous streams of air and exhaust gas are deleted, replaced with semi-continuous counterparts consisting of the appropriate phases. New instances of the discrete streams presented in Table 8.2 are introduced, consisting of the specified phases and specified according the association process used earlier to specify the constituents of the originally continuous streams. The resulting streams layer is depicted in Figure 8.6.

8.6 Technique Selection

The practitioner searches for appropriate techniques listed centrally by the Ontology Repository, to mathematically represent concept instances created in the layered representation. The Technique repositories that publish the relevant techniques provide information describing the numerical variables that must be specified to initialise each technique. The practitioner then specifies the numerical values for these variables in the user interface. The simulation is subsequently commenced to generate the time-varying results for the problem.

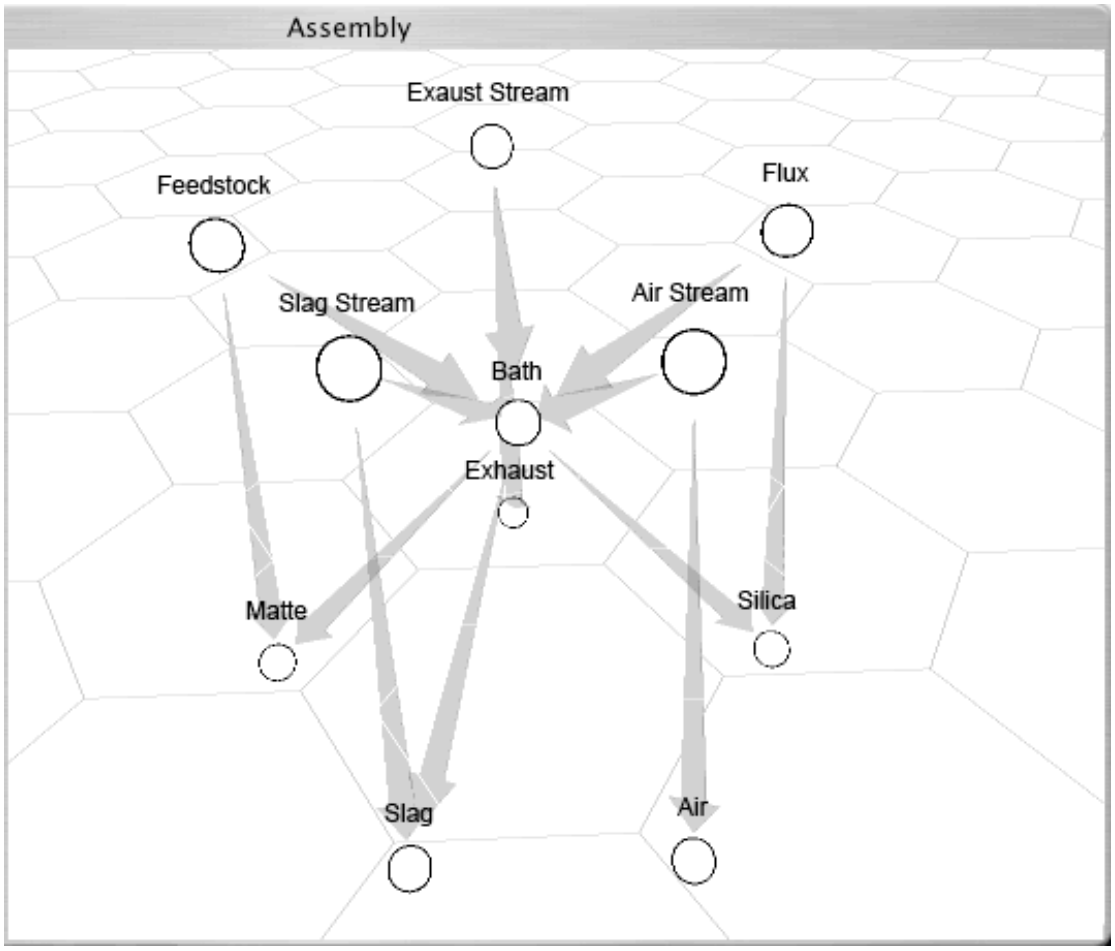


Figure 8.6: Screenshot depicting the streams constructed in the streams layer.

8.7 Summary

This chapter has demonstrated the use of the implementation through application to a simplified process-engineering problem. Assembly of the problem involved (a) creation of concept instances within the appropriate layers (b) connection of concept instances to relevant concept instances in lower layers, and (c) the choice of remote techniques instances used to represent concept instances. The next chapter presents the conclusion to this thesis.

Chapter 9: Conclusions

This thesis has analysed, designed, and implemented an ontology-based simulation-assembly infrastructure around the separation of roles of practitioner, developer, and architect. The resulting system demonstrated how we can partition the resulting software system into respective components so that each of role can work within their primary skillset, to reliably contribute to the process of assembling simulations. In this way, we developed a system that shows (a) practitioners do not have to be literate in text-based programming languages—a secondary or peripheral skill to these specialists—to assemble simulations, (b) practitioners can have access to more ontologies so as to reconfigure and extend their simulation-assembly environment, should that environment be immediately inadequate to their particular problem domain, and (c) practitioners can be removed from the software development process involved in writing reliable and tested mathematical techniques, lowering the cognitive burden upon the practitioner by allowing them to configure existing building blocks within their environment. In this way, we can maximise the confidence in the results produced from simulation.

We showed that it was more intuitive to use 3D layers for visualising the conceptual assembly of simulation problems. The implementation demonstrates the feasibility of presenting complex concept sets consisting of any number of concepts in a form accessible to the practitioner to (a) navigate and create concept instances in each layer and (b) to provide extensibility of the environment to arbitrary ontologies.

We found that the building block mathematical techniques underpinning the resulting system must have an accompanying visual definition of an associated dialogbox, to configure those techniques associated with concept instances within the practitioner's runtime graphical environment. This will provide a useful area of research in future work for addition to the software system.

We found that recent technologies can now enable the greater interoperability of mathematical techniques written in multiple programming languages. Although not included in the current implemetnation, we rationalised that future incorporation of the intermediate language representation (IR) facilitated by the LLVM Compiler Infrastructure—see http://llvm.org—will alleviate the issue of selecting a canonical programming language for use by developers to implement their mathematical techniques—thus removing the use of GCC C++ share libraries within the current implementation presented in this thesis. This permits developers to implement their mathematical techniques in arbitrary, commodity programming languages, storing the resulting universally interoperable LLVM IR code from these languages in repositories for mathematical techniques.

During the work, it was demonstrated how an example domain ontology could be designed, specified, and incorporated into the architecture—in this case for the process-engineering domain. We went onto outline a simplified example simulation problem for this domain to provide the basis to an example walkthrough of how a practitioner assembles simulations within the layered visualisation approach.

The analysis, design, specification, and implementation presented in this study produced the following conclusions regarding the design of simulation software. By clearly separating the compile-time and run-time goals of simulation assembly, we can influence the means by which practitioners interact with software to express simulations, removing the burden from their day-to-day work of managing a potentially large code base written in yet-another modelling language.

The separation of compile-time goals and run-time goals affects the physical and logical design of the related software, shifting the definition of domain ontologies into the hands of the architect role. In this way, the foremost thinkers of a field can centralise the related concept set for that domain. Practitioners can then draw upon these standardised ontologies rather than attempting to redefine those concept types. This separation also shifts the implementation of mathematical techniques into the hands of the developer role, who can use commodity programming languages for this task. In this manner, a much larger developer base is available to create, maintain, and document these mathematical techniques.

By shifting from a symbolic graphical notation of concept instances to that of a node-based referential notation of concept instances, we can extend the notion of the classical flowsheet—involving two levels of hierarchical decomposition—into a multi-layered depiction of run-time concept instances. This approach provides a run-time means of depicting simulation problems involving more than two concept types, so that compile-time steps can be removed from the practitioner's simulation environment. The practitioner no longer needs a text-based programming language to express concepts of a granularity finer than that presented in the related flowsheet.

This node-based depiction is more generic than any flowsheet. Instances of concept types are visually created in a specific layer related to the concept type: the layer rather than the symbol differentiates the concept type of the related instances. It follows that there is no constraint of needing to invent a proliferation of symbols to distinguish each concept type, and subsequently any arbitrary domain ontology can be used as a basis to the layer ordering. In other words, this form of visualisation can be applied to any discipline or field—not just process engineering. The proviso is that the domain ontology defining the concept set which orders the layers of the representation, must involve directed and acyclic relationships amongst those concept types, so as to work with this layer system.

9.1 Future Work and Recommendations

This thesis has provided an approach for managing the interaction by practitioners with an ontology-based, simulation-assembly infrastructure. While the foundational software components have been implemented to show proof of concept, further facilities are required to better realise the overall system, including incorporation of LLVM IR for technique implementation, and the need to couple dialogbox-definition information into the techniques written by the developer role, for storage in the technique repository. Furthermore, as it is beyond the scope of this PhD to perform a complete usability study investigating the demographics of practitioner responses to the softare approach developed herein, the author recommends that just such a usability study be performed in future work.

References

1. Aho, A. V., and Ullman, J. D., “Principles of Compiler Design”, Addison-Wesley Publishing Company, 1977.

2. Allan, B. A., Armstrong, R. C., Wolfe, A. P., Ray, J., Bernholdt, D. E., and Kohl, J. A., “The CCA Core Specification In A Distributed Memory SPMD Framework”, Concurrency and Computation: Practice and Experience, v14, n5, 2002, pp323-345. See also http://www.cca-forum.org.

3. Andersson, M., "Omola—An Object-Oriented Language for Model Representation", Licentiate Thesis, Department of Automatic Control, Lund University of Technology, Sweden, 1990.

4. AspenTech Pty. Ltd., 2003. See http://www.aspentech.com.

5. Bateman, I. R., “Mathematical Modelling and Optimal Control of a Copper Anode Furnace”, St. Lucia, Qld., 1992.

6. Beck. K., “Extreme Programming Explained: Embrace the Change”, Addison-Wesley Publishing Co.; 1st edition, October 1999.

7. Bogusch, R., Lohmann, B., Marquardt, W., “Computer-Aided Process Modelling with MODKIT”, Computers and Chemical Engineering, v25, 2001, pp963-995.

8. Booch, G., “Object-Oriented Analysis and Design with Applications”, Addison-Wesley, 2nd Edition, 1994.

9. Braunschweig, B., Fraga, E. S., Guessoum, Z., Paen, D., Pinol, D., Yang, A., “COGENTS: Cognitive Middleware Agents to Support E-CAPE”, E-WORK 2002 Conference, Prague, October 2002.

10. Brickley, D., Guha, R. V. (Eds.), “RDF Vocabulary Description Language 1.0: RDF Schema”, W3C Working Draft, January 2003. See http://www.w3.org/TR/rdf-schema/.

11. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M., “Pattern-Oriented Software Architecture, Volume 1: A System of Patterns”, John Wiley & Son Ltd,, 1st Edition, 1996.

12. CACI Products Company “CACI. MODSIM III”, La Jolla, CA, 1997. See http://www.caciasl.com/modsim.html.

13. Chandrasekaran, B., "Generic Tasks in Knowledge-Based Reasoning: High-Level Building Blocks for Expert Systems Design", IEEE Expert, v1, n3, 1986, pp23-30.

14. CO-LaN: The CAPE-OPEN Laboratories Network, 2003. See http://www.colan.org.

15. COGents website, 2003. See http://www.cogents.org.

16. Cubert, R. M. and Fishwick, P. A., “Sim++”, Version 1.0. Department of Computer and Information Science and Engineering, University of Florida, Gainesville, Florida, 1995. See http://www.cis.ufl.edu/~fishwick.

17. Cubert, R. M., and P. A. Fishwick. “MOOSE: An Object-Oriented Multimodelling and Simulation Application Framework”, Simulation, v70, n6, 1998, pp379-395.

18. Dhal, O.,-J. and Nygaard, K., "SIMULA—An Algol-based Simulation Language," Communications of the ACM, v9, n9, September 1966, pp671-678.

19. Daesim Technologies, 2003. See http://www.daesim.com.

20. Dijkstra, E. W., “The Structure of the THE-Multiprogramming System.”, Communications of the ACM, v1, n5, May 1968, pp341-346.

21. D’Souza, D. F., “Types and Classes: A Language-Independent View”, Journal of Object-Oriented Programming (JOOP), v10, n1, 1997, pp 10-13.

22. D'Souza, D. F., and Wills, A. C., "Objects, Components, and Frameworks with UML; The Catalysis Approach", Addison-Wesley Object Technology Series, 1999.

23. Duke, R., Rose, G., and Smith, G., "Object-Z: A Specification Language Advocated for the Description of Standards", Computer Standards and Interfaces, v17, pp511-533, September 1995.

24. Eddon, G. and Eddon, H., "Inside Distributed COM", Microsoft Press, April 1998. See also http://www.microsoft.com/com/.

25. Elmqvist, H., "Dymola—Dynamic Modelling Language User's Manual" Dynasim AB, Lund, Sweden, 1993. See http://www.dynasim.com.

26. Fagan, M. E., "Design and Code Inspections to Reduce Errors in Program Development", IBM Systems Journal, v15, n3, pp182-211. 1976.

27. Fiedler, S. P., "Object-Oriented Unit Testing," Hewlett-Packard Journal, v36, n4, April 1989, pp69 - 74.

28. Foster, I. and Kesselman, C. (eds.), "The Grid: Blueprint for a New Computing Infrastructure", Morgan Kaufmann Inc., San Francisco, CA, 1998.

29. Fowler, M. and Scott, K., "UML Distilled: A Brief Guide to the Standard Object Modelling Language", Addison-Wesley Publishing Co., 1999.

30. Gamma, E., Helm, R., Johnson, R., and Vlissides, J., "Design Patterns", Addison-Wesley, 1st Edition, 1995.

31. Heineman, G. T. and Councill, W. T., "Component-Based Software Engineering: Putting the Pieces Together", Addison-Wesley Pub. Co., June 2001.

32. Hayes-Roth, F., Lenat, D., and Waterman, D., (eds.), "Building Expert Systems", Addison-Wesley, Reading, Massachusetts, 1983.

33. Humphrey, W. S., “A Discipline of Software Engineering”, Addison-Wesley, Reading, Massachusetts, 1995.

34. Hyprotech Pty. Ltd., 2003. See http://www.hyprotech.net.

35. International Business Machines (IBM) and Object Technology International (OTI), “Eclipse Platform Technical Overview v2.1”, Object Technology International Inc., February 2003. See http://www.eclipse.org, http://www.ibm.com, and http://www.oti.com.

36. Internet Engineering Task Force (IETF), “Functional Requirements for Uniform Resource Names”, IETF, RFC1737, December 1994. See http://www.ietf.org.

37. Jacobson, I. and Christerson, M., "A Confused World of OOA and OOD", Journal of Object-Oriented Programming (JOOP), September 1995. pp15-20.

38. Jensen, A. K. and Gani, R., “Generation of Problem Specific Simulation Models within an Integrated Computer-Aided System”, PhD thesis, Department of Chemical Engineering, DTU, Denmark, 1998.

39. Jensen, F. V., “An Introduction to Bayesian Networks”, Springer-Verlag, New York, 1996.

40. Jennings, N. P. and Wooldridge, M. J. (eds.) “Agent Technology: Foundations, Applications, And Markets”, Springer, Berlin Heidelberg, New York, 1998.

41. Keahey, K. and Gannon, D., “PARDIS: CORBA-based Architecture for Application-Level Parallel Distributed Computation”, Proceedings of Supercomputing ‘97”, November 1997.

42. Kidd, A. L. (Ed.), “Knowledge Acquisition for Expert Systems”, Plenum, New York, 1987.

43. Kravanja, Z., and Grossmann, I. E., “New Development and Capabilities in PROSYN—An Automated Topology and Parameter Process Synthesizer”. Computers Chem. Engng., v18, n11/12, 1994, pp1097-1114.

44. Lenat, D. B. and Guha, R. V., “Building Large Knowledge-Based Systems: Representation and Inference in the Cyc Project”, Addison-Wesley Publishing Co., February 1990.

45. Martin, R. C., “Design Principles and Design Patterns”, Object Mentor Inc., 2000. See http://www.objectmentor.com.

46. The MathWorks Inc., Natick, “MATLAB Reference Guide”, August 1992. See http://www.mathworks.com.

47. Mattsson, S.E., Elmqvist, H., Broenink, J.F., “Modelica: An International Effort to Design the Next Generation Modelling Language", Journal A—Benelux Quarterly Journal on Automatic Control, v38, n3, September 1997. pp16-19. See also http://www.modelica.org

48. McNab, R. and Howell, F.W., ``Using Java for Discrete Event Simulation" in proc. Twelfth UK Computer and Telecommunications Performance Engineering Workshop (UKPEW), Univ. of Edinburgh, 1996, pp219-228.

49. The Modelica Association, “Modelica—A Unified Object-Oriented Language for Physical Systems Modelling: Language Specification Version 1.4”, December 2000. See http://www.modelica.org.

50. Musen, M. A., “Ontology-Oriented Design and Programming”, In: Cuena, J., Demazeau, Y., Garcia, A., and Treur, J., Knowledge Engineering and Agent Technology. Amsterdam: IOS Press (in press), 2001.

51. Object Management Group “The Common Object Request Broker: Architecture and Specification”, February 1998. See http://www.omg.org/corba/.

52. Object Management Group “Meta Object Facility (MOF) Specification v1.3”, March 2000. See http://www.omg.org/technology/cwm/.

53. Object Management Group, “OMG Unified Modelling Language Specification”, September 2001. See http://www.omg.org/uml/.

54. Object Management Group “OMG XML Metadata Interchange (XMI) Specification v1.2”, January 2002. See http://www.omg.org/technology/xml/.

55. Opdyke, W. F., “Refactoring Object-Oriented Frameworks”, PhD thesis, University of Illinois, Urbana-Champaign, 1992.

56. Page-Jones, M., “Fundamentals of Object-Oriented Design in UML”, Addison-Wesley Pub. Co., 2000.

57. Piela, P., “ASCEND: An Object-Oriented Computer Environment for Modelling and Analysis”, Ph.D. Thesis, Carnegie Mellon University, Pittsburgh, PA, 1989.

58. Piela, P., Epperly, T., Westerberg, K., and Westerberg, A., “ASCEND: An Object-Oriented Computer Environment for Modelling and Analysis: The Modelling Language.” Computers and Chemical Engineering, v15, n1, 1991. pp53-72.

59. Platt, D. S., “Introducing Microsoft .Net”, Microsoft Press, 3rd edition, April 2003. See also http://www.microsoft.com/net/.

60. Process Systems Enterprise Ltd., “gPROMS: Advanced Users Guide”, London, UK, Process Systems Enterprise Ltd. See http://www.psenterprise.com.

61. ProSim Pty Ltd (2003). See http://www.prosim.net

62. Qian, Y., Huang, Q., Lin, W., Li, X., “An Object/Agent Based Environment for the Computer Integrated Process Operation System”, Computers and Chemical Engineering, v24, 2000. pp457-462.

63. Riel, A. J., “Object-Oriented Design Heuristics”, Addison-Wesley, 1st Edition, 1996.

64. Rumbaugh, J., Blaha, M., Premerliani, W., Eddy, F., Lorensen, W., "Object-Oriented Modelling and Design", Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

65. Williams, R., Hangos, K., McGahey, S., Cameron, I., “Assumption Based Modelling and Mode Documentation”, In 6th World Congress of Chemical Engineering, September 2001.

66. Scott, M. L., “Programming Language Pragmatics”, Morgan Kaufmann Publishers, San Francisco, California, 2000.

67. Stephanopoulos, G., Johnston, J., Kriticos, T., Lakshmanan, R., Mavrovouniotis, M. L., and Siletti, C., “Design-Kit: An Object-Oriented Environment for Process Engineering.”, Computers and Chemical Engineering, v11, n6, 1987. pp655-674.

68. Stephanopoulos, G., Henning, G., and Leone, H., "MODEL.LA: A Modelling Language for Process Engineering, Parts I-II", Computers and Chemical Engineering, v14, n8, 1990. pp813-869.

69. Sun Microsystems, “Java Remote Method Invocation Specification, v1.4”, Sun Microsystems, 2003. See http://java.sun.com/products/jdk/rmi/.

70. Sun Microsystems, “Java 2 Platform Enterprise Edition Specification v1.4”, Sun Microsystems, 2003. See http://java.sun.com/j2ee/.

71. Szyperski, C., “Component Software: Beyond Object-Oriented Programming”, Addison-Wesley Pub. Co., December 1997.

72. van Harmelen, F., Patel-Schneider, P.F. (Eds.), Horrocks, I., “Reference Description of the DAML+OIL Ontology Markup Language v4.2”, March 2001. See http://www.cs.vu.nl/~frankh/.

73. van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D. L., Patel-Schneider, P. F., Stein, L. A., “OWL Web Ontology Language Reference”, W3C Working Draft, March 2003. See http://www.w3.org/TR/owl-ref/.

74. VEDA Team, “The Chemical Engineering Data Model VEDA: Parts 1-6”, Technical Report, Lehr- und Forschungsgebiet Theoretische Informatik, RWTH Aachen. 1998.

75. v. Wedel, L. and Marquardt, W., “ROME: A Repository to Support the Integration of Models over the Lifecycle of Model-based Engineering Processes”, European Symposium on Computer Aided Process Engineering-10, Pierucci, S. (Ed.), Elsevier, 2000, pp535-540.

76. Weitzenfeld, A., Arbib, M., and Alexander, A., “The NSL Book: Script Language”, MIT Press, September 1999.

77. White, J. E., “Mobile Agents”, in Software Agents, J. Bradshaw (Ed.),. AAAI Press/MIT Press, Menlo Park, 1997. pp437-472.

78. Wilkes, M. V., D. J. Wheeler, D. J, and Gill, S., “The Preparation of Programs for an Electronic Digital Computer”, Addison-Wesley Publishing. Co., Cambridge Massachusetts, 1951.